

12/14/99



JC715 U.S. PTO

Frederick P. Fish
1855-1930

W.K. Richardson
1859-1951

FISH & RICHARDSON P.C.

12-16-99

December 14, 1999

4225 Executive Square
Suite 1400
La Jolla, California
92037

Telephone
858 678-5070

Facsimile
858 678-5099

Web Site
www.fr.com

JC542 U.S. PTO
09/461160



Attorney Docket No.: 06618/389001/CIT-3102

Box Patent Application

Assistant Commissioner for Patents
Washington, DC 20231

Presented for filing is a new patent application claiming priority from a provisional patent application of:

Applicant: JOHN THORNLEY, K. MANI CHANDY AND HIROSHI ISHII

Title: PROGRAMMING SYSTEM AND THREAD SYNCHRONIZATION
MECHANISMS FOR THE DEVELOPMENT OF SELECTIVELY
SEQUENTIAL AND MULTITHREADED COMPUTER PROGRAMS

Enclosed are the following papers, including those required to receive a filing date under 37 CFR 1.53(b):

	<u>Pages</u>
Specification	57
Claims	17
Abstract	1
Declaration	[To be Filed at a Later Date]
Drawing(s)	4

Enclosures:

- Computer program appendix, 27 pages.
- Postcard.

CERTIFICATE OF MAILING BY EXPRESS MAIL

Express Mail Label No. EL528179775US

I hereby certify under 37 CFR §1.10 that this correspondence is being deposited with the United States Postal Service as Express Mail Post Office to Addressee with sufficient postage on the date indicated below and is addressed to the Assistant Commissioner for Patents, Washington, D.C. 20231.

Date of Deposit December 14, 1999

Signature Deborah Dean

Typed or Printed Name of Person Signing Certificate

BOSTON

DELAWARE

NEW YORK

SILICON VALLEY

SOUTHERN CALIFORNIA

TWIN CITIES

WASHINGTON, DC

FISH & RICHARDSON P.C.

Assistant Commissioner for Patents

December 14, 1999

Page 2

Under 35 USC §119(e)(1), this application claims the benefit of prior U.S. provisional application 60/112,817, filed December 17, 1998.

This application is entitled to small entity status. A small entity statement will be filed at a later date.

Basic filing fee	\$0
Total claims in excess of 20 times \$9	\$0
Independent claims in excess of 3 times \$39	\$0
Fee for multiple dependent claims	\$0
Total filing fee:	\$0

No filing fee is being paid at this time. Please apply any other required fees, **EXCEPT FOR THE FILING FEE**, to deposit account 06-1050, referencing the attorney document number shown above. A duplicate copy of this transmittal letter is attached.

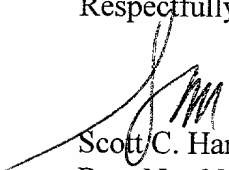
If this application is found to be incomplete, or if a telephone conference would otherwise be helpful, please call the undersigned at (858) 678-5070.

Kindly acknowledge receipt of this application by returning the enclosed postcard.

Please send all correspondence to:

SCOTT C. HARRIS
Fish & Richardson P.C.
4225 Executive Square, Suite 1400
La Jolla, CA 92037

Respectfully submitted,


Scott C. Harris
Reg. No. 32,030
Enclosures
SCH/jzc
10010834.doc

APPLICATION
FOR
UNITED STATES LETTERS PATENT

TITLE: PROGRAMMING SYSTEM AND THREAD
SYNCHRONIZATION MECHANISMS FOR THE
DEVELOPMENT OF SELECTIVELY SEQUENTIAL AND
MULTITHREADED COMPUTER PROGRAMS

APPLICANT: JOHN THORNLEY, K. MANI CHANDY AND HIROSHI
ISHII

CERTIFICATE OF MAILING BY EXPRESS MAIL

Express Mail Label No. EL528179775US

I hereby certify under 37 CFR §1.10 that this correspondence is being deposited with the United States Postal Service as Express Mail Post Office to Addressee with sufficient postage on the date indicated below and is addressed to the Assistant Commissioner for Patents, Washington, D.C. 20231.

Date of Deposit December 14, 1999

Signature

Laborah Dean
Typed or Printed Name of Person Signing Certificate

**PROGRAMMING SYSTEM AND THREAD SYNCHRONIZATION MECHANISMS FOR THE
DEVELOPMENT OF SELECTIVELY
SEQUENTIAL AND MULTITHREADED COMPUTER PROGRAMS**

5

The present application claims priority under 35 U.S.C.
119(e) from provisional application number 60/112,817 filed
December 17, 1998.

10

Background

15

20

25

Many computer programs are computationally intensive,
meaning that they require large amounts of computing power. As
a consequence, these programs may execute more slowly than the
computer user desires, even on the fastest computers. One way
of increasing the execution speed of a computationally intensive
computer program is to divide the program into multiple units,
or loci, of concurrent execution. These units of execution are
known as "threads". A program with multiple threads of
execution is known as a "multithreaded program". A program with
only a single thread of execution is known as a "sequential
program". The threads that make up a multithreaded program may
be executed concurrently on multiple computer processors,
allowing many operations in the program to be carried out
simultaneously, thereby speeding up program execution.

In a multithreaded program, the program or operating system must control the access of threads to data objects in the program, in order to prevent the multiple threads from concurrently accessing the same data object in an undesirable manner. If multiple threads modify the same data object concurrently, or read and modify the same data object concurrently, the resulting state of the program is extremely difficult to determine. Developing a multithreaded program is significantly more difficult than developing a sequential program because of the problems of (1) expressing the division of a program into multiple threads and (2) structuring and controlling the access of those threads to data objects.

Summary

The present application teaches a structured thread ("Sthread") system with thread synchronization and production mechanisms.

Another aspect produces multithreadable code. The multithreadable code can be annotated using information indicative of its multithreadability. The multithreadable code constructs are code constructs that can be executed in a multithreadable manner, or equivalently in a sequential manner. Multithreadable code constructs may be expressed by annotating sequential code constructs to indicate that their multithreaded

execution is equivalent to sequential execution. The multithreadable code can be used along with multithreaded code. Specific instances of a multithreadable constructs: a multithreadable block, and a multithreadable for loop, and are disclosed.

The second aspect of the system is the integration of multithreadable code constructs with traditional explicitly multithreaded code constructs. Explicitly multithreaded code constructs must always be executed in a multithreaded manner. Examples of explicitly multithreaded code constructs include multithreaded block constructs, multithreaded for loop constructs, and library-based thread creation functions. Multithreadable code constructs and explicitly multithreaded code constructs may be intermingled within a program as required, with well-defined meaning.

According to a first aspect, a special counter called an "s-counter", is used as a thread synchronization mechanism. Special "s-Flags" can also be used for thread synchronization, and flag synchronization is also described herein.

Yet another aspect is the implementation of the programming system within an existing compiler environment using a special pre-processing system.

The embodiments of the invention describe additional details, including the following:

The s-counter synchronizing the access of threads to shared data objects. The mechanisms use "monotonic" synchronization objects, with operations that can be constrained to only move the value of the object in one direction. Monotonic
5 synchronization objects can be used to synchronize the access of threads to shared data objects in multithreadable code constructs in a manner that guarantees the equivalence of sequential and multithreaded execution. Specific instances of monotonic synchronization objects are disclosed, namely a form
10 of counter called an "s-counter" and a form of flag called an "s-flag". The s-counter is a particularly powerful thread synchronization mechanism in many contexts, with its use in multithreadable code constructs being one example.

The application describes implementation of the
5 multithreaded programming system within an existing program development and compilation environment using a special source-to-source preprocessing system and high-level thread library. This allows the system to be transparently and seamlessly integrated with existing programming systems such as Microsoft
20 Visual Studio for the Microsoft Windows family of operating systems, the GNU program development tools for Linux and other versions of the Unix operating system, or on any version of the Java programming language, for example.

Brief Description Of The Drawings

These and other aspects will now be described in detail with respect to the accompanying drawings, wherein:

Figure 1 is a process flowchart showing a prior art method
5 for compiling multithreaded code;

Figure 2 shows a computer system and its thread allocation system;

Figure 3 shows a flowchart of defining an s-counter; and

10 Figure 4 shows a flowchart of Sthreads execution of a program.

Detailed Description

FIG. 1 is a process flowchart showing a prior art method for compiling multithreaded code. Source code text 300 including
15 multithreaded code constructs is processed by a conventional compiler 302. The compiler communicates with a linker 304 which links pre-existing routines from a library 306 with the output of the compiler to create an executable module 308.

Existing operating systems, including the WIN32 API, often
20 provide a general purpose thread library which may allow carrying out defined tasks like these. For example, a first thread may be defined for operating the CD ROM, and another for the modem.

Windows NT WIN32 thread creation is unstructured. A thread is created by passing a function pointer and an argument pointer to a CREATETHREAD call. The new thread then executes the given function with the given argument. There is no specific
5 relationship between the created thread and the creating thread: the two threads are effectively asynchronous. One thread for example, can arbitrarily suspend, resume or terminate the execution of another thread.

This is not a problem for unrelated tasks like CD/modem
10 tasks noted above. However, when two parts of a program are to be executed as threads, the synchronization operations are often complex and error prone. Unpredictable interactions among the multiple threads can induce problems including race conditions, and deadlock. Effectively, the user is left with the daunting
5 task of using these thread libraries in a way that does not cause this problem.

The present application discloses a specific embodiment operating using the Windows NT (TM) system. It should be understood, however, that this system is portable across many
20 platforms and that the same concept described herein can be used in those systems, including Linux, and any other operating system.

While a process has its own address space, a thread is often simply a program counter and stack pointer. A process may

have many threads but all the threads share the same address space.

Figures 2 and 3 show this operation in a computer system. Figure 2 shows a computer system, with four processors 200, 202, 204, 206. The processors can be in a multiple processor system as shown. The pool 199 of processors is associated with an operating system 210, a user interface 215, auxiliary hardware 220 (e.g. memory, chipsets, etc), a display 225 and other computer components.

The operating system 210 includes multiple threads 212, 214, and others. Each thread is resident on the stack within the heap. The threads are associated with processors, which execute the threads. Figure 2 shows the pool of threads on the left and the pool of processors on the right. Each of the dynamically-created threads are peers. Of course there can be many more threads than processors. The operating system controls the threads to dynamically switch between the processors.

The present application defines an entirely new way of creating, synchronizing, and handling the synchronization among threads. The system uses a new way of compiling code based on multithreadable code, either alone, or in conjunction with multithreaded code.

The operating system or programming language has a higher level system that uses special constructs called "equivalency annotations". A lower level function call based system is used with special objects. Those special objects can synchronize
5 among the threads in a way that prevents the objects within the threads from having ambiguous states.

Many of these systems are based on the concept of equivalence annotations. Equivalence annotations can take many forms - pragmas, special keywords, special kinds of comments, special characters, textual modifications (such as boldfacing, underlining, or italics), and others. They could be part of the program text, or in a separate file i.e., an extra file that contains nothing but the annotations. The pragma can have meaning to a compiler. Pragmas often form a specified syntax, but usually convey nonessential information that is intended to help the compiler to optimize the program.
10
5

The present embodiment uses these pragmas as special equivalence annotations. Pragmas are convenient for annotations since many programming language already provide pragmas for
20 other purposes. While a pragma is described as being used as the preferred annotation of the present application, the program can certainly be annotated in other manners. For example, Java does not support pragmas, so a special kind of comment line could be used. The equivalence annotations described throughout

this specification should be understood to be interpretable in this way.

The MULTITHREADABLE equivalence annotation can be a pragma when embodied in the C programming language. This indicates that a block or loop can be executed in a multithreaded manner. This means that there is no timing dependent nondeterminacy, and the system can execute the instructions into a multithreaded system.

The MULTITHREADED equivalence annotation means that a block or loop must be executed in a multithreaded manner. The multithreaded execution need not be equivalent to sequential execution. Lock synchronization can be used to introduce nondeterminacy if desired.

The equivalence annotation becomes part of the operating system. Special, monotonically increasing and otherwise constrained s-COUNTERS, and similarly constrained s-FLAGS are operated to synchronize the access of threads to shared memory in order to prevent unwanted interference.

A special SYNCHRONIZATION COUNTER, or s-COUNTER is defined as an object with three basic attributes. The s-COUNTER has a non-negative integer value. The object only allows an INCREMENT operation and a CHECK operation. An initial value of the s-COUNTER object is set to zero. An INCREMENT function automatically increases the value of the counter by a specified amount. The

CHECK operation suspends the calling thread until the value of the counter becomes greater than or equal to a specified level.

The multi-threaded programming system has a higher level notation includes annotation objects in the program code. Using the example of the c language, this can be described as "multi-threaded c".

A lower level structured thread library is described as "Sthreads". The special annotation objects are transformed into special Sthread calls by the Sthreads annotation objects pre-processor.

The multithreaded model uses the thread synchronization/annotation objects disclosed above to synchronize among the threads.

Threads can be created in different ways. A first thread creation construct is the multithreaded block. This is indicated by the MULTITHREADED keyword placed immediately before an ordinary C block:

```
MULTITHREADED {
    statement
    ...
    statement
}
```

This notation specifies that the statements of the block should be executed as asynchronous threads. This is a conventional way of referring to these threads. For example, the operating system could create threads to read from CD, and

threads to read from tape. The threads are executed and proceed concurrently. They all share the same address space as the parent program. Execution does not continue past the multithreaded block until all the threads have individually terminated. It is typically illegal for the program to contain any kind of jump between the individual statements of the block, from inside the block to outside the block, or from outside the block to inside the block.

A second thread creation construct is the multithreaded for-loop, indicated by the MULTITHREADED keyword placed immediately before an ordinary for-loop:

```
MULTITHREADED
FOR (I = expression; I comparison expression; I = I + expression)
    statement
```

This notation specifies that the iterations of the loop should be executed as asynchronous threads. The threads all share the same address space as the parent program. Each thread, however, has a local copy of the loop control-variable with a different value from the iteration range. The iteration scheme can restrict to a single control-variable and expressions that are not modified within the loop body. Execution does not continue past the multithreaded for-loop until all the threads have individually terminated. It is illegal for the program to contain any kind of jump from inside the loop to outside the loop or from outside the loop to inside the loop. In essence, a

multithreaded for-loop is a quantified form of multithreaded block.

Multithreaded and ordinary blocks and for-loops can be arbitrarily nested.

5 Traditional approaches have often been categorized as either being explicitly multithreaded or implicitly multithreaded. With explicit multithreading, the programmer expresses exactly how the operations of the program are executed by threads. Implicit multithreading is carried out when the
10 programmer writes an ordinary sequential program. The programming system, e.g. the compiler, determines how the operations can be executed by separate threads.

 The present application goes beyond the multithreaded concepts described above into a concept of multithreadable code
5 constructs. The multithreadable construct can be executed according to a specified sequential operational semantics. The most common operational semantics would be executed sequentially. An alternative, however, allows the multithreadable code construct to be operated according to
20 multithreaded operational semantics.

 Rules are defined that constrain the multithreaded execution such that its result is equivalent to sequential execution.

As disclosed herein, the multithreadable code construct is formed of:

- i) a syntactic description of the form of the construct,
- (ii) a sequential operational semantics, that, when
5 executed, defines how to execute the construct sequentially,
- (iii) a multithreaded operational semantics, defining how
to execute the construct by a set of threads, and
- (iv) a set of implicit or explicit programming rules that
are sufficient to ensure the equivalence of sequential and
10 multithreaded execution of the construct.

The MULTITHREADABLE pragma becomes an assertion by the programmer that the BLOCK or FOR loop can be executed in a multithreaded manner without changing the results of the program. The MULTITHREADABLE pragma can be applied to blocks and for
5 loops in which the statements or iterations are independent of each other. The multithreaded execution is equivalent to sequential execution in this case. It is not a directive that the BLOCK or FOR LOOP *must* be executed in a multithreaded manner.

As a simple example, consider the following program to sum
20 the elements of a two-dimensional array:

```
void SumElements(float A[N][N], float *sum, int numThreads)
{
    int i;
    float rowSum[N];
    25
    #pragma multithreadable mapping(blocked(numThreads))
    for (i = 0; i < N; i++) {
        int j;
```

```

    rowSum[i] = 0.0;
    for (j = 0; j < N; j++)
        rowSum[i] = rowSum[i] + A[i][j];
}
*sum = 0.0;
for (i = 0; i < N; i++)
    *sum = *sum + rowSum[i];
}

```

Multithreaded execution of the FOR loop is equivalent to sequential execution because the iterations all modify different ROWSUM[i] and j variables. The arguments following the pragma indicate that multithreaded execution should assign iterations to NUMTHREADS different threads using a blocked mapping. There is a rich set of options that control the mapping of iterations to threads.

Therefore, the Multithreaded C preprocessor has two modes: sequential mode in which the MULTITHREADABLE pragma is ignored, and multithreaded mode in which the MULTITHREADABLE pragma is transformed into Sthreads calls. Programs can be developed, tested, and debugged in sequential mode, then executed in multithreaded mode for performance. In addition, performance analysis and tuning can often be performed in sequential mode.

Determinacy of results is an important consequence of the equivalence of multithreaded and sequential execution. If sequential execution is deterministic (which is usually the case), multithreaded execution must also be deterministic. Determinacy is usually desirable, since program development and debugging can be difficult when different runs produce different

results. In other multithreaded programming systems, determinacy is difficult to ensure. For example, locks, semaphores, and many-to-one message passing almost always introduce race conditions and hence nondeterminacy. However, nondeterminacy is
5 important for efficiency in some algorithms, e.g., branch-and-bound algorithms.

Multithreaded and multithreadable code constructs are integrated in this system. The programming system incorporates both explicitly multithreaded constructs which must be executed
10 according to the multithreaded semantics, along with multithreadable constructs which may selectively executed according to their sequential or multithreaded semantics. The multithreaded constructs are generally used to express multithreaded algorithms that have no sequential equivalent.
15 This can include controlling different hardware that haw no integration with one another, or controlling simultaneous different windows in a graphical user interface.

Multithreadable constructs are used to express the opportunity to use multithreading to speed up the execution of a
20 computationally-intensive algorithm, by using multiple threads on multiple processors.

By using both multithreaded and multithreadable constructs, the operating system can use one thread to control each window with multithreaded constructs, and the output to each window,

within a window, is computed faster with the multiple threads using multithreadable constructs.

As described above, the synchronization can be carried out by an entry s-COUNTER, or an s-FLAG. Each are defined to have
5 certain constraints.

An S-COUNTER, defined in the context of the C programming language, is diagrammed in Figure 3. It can be defined as a type definition and a set of interface functions. The counters are encapsulated as a class in an object-oriented language such
10 as C++ or Java. The definition of the fundamental programming interface for S-COUNTERS is as follows:

```
typedef counter type definition Counter;
int InitializeCounter(Counter *c);
/* Initializes value(c) to zero. */
/* Must be called only once, before all other operations on counter c. */
int FinalizeCounter(Counter *c);
/* Must be called only once, after all other operations on c. */
int CheckCounter(Counter *c, unsigned int level);
/* Suspends until value(c) greaterorequal level. */
int IncrementCounter(Counter *c, unsigned int amount);
/* Increases value(c) by amount. */
```

An s-COUNTER object c implicitly has a nonnegative integer attribute value(c), which can only be accessed through the interface functions. The INITIALIZE function at 300 initializes
30 value(c) to zero or some initial value.

Importantly, the counter is monotonic, as illustrated in
310. No decrement function is defined. Its value monotonically increases.

Any attempt to CHECK the counter, shown as step 320, suspends the calling thread at 325. This prevents a condition which can catch or miss some action occurring during the check operation. Each s-COUNTER maintains a dynamic list of thread suspension queues 330, with one queue for each value on which at least one Check operation is suspended.

CHECK compares value(c) to level and suspends until value(c) becomes greater than or equal to level. This is generically shown as AWAKE in step 340. Increment at 310 atomically increases value(c) by amount, thereby reawakening all Check operations suspended on values less than or equal to the new value(c).

All the functions can return an error code. Possible error conditions include invalid arguments, operations on an uninitialized counter, and counter overflow.

The type definition described above is carefully selected to remove the possibility of race conditions occurring on counter synchronization. There is no DECREASE operation.

Therefore, the value of an s-COUNTER is monotonically increasing.

There is no possibility of a CHECK operation missing an INCREMENT operation since check suspends the thread. There is no PROBE or nonblocking CHECK operation. It is recognized by the inventor that any instantaneous value may depend on the relative timing

of the individual threads. Therefore, no operation can be based on the instantaneous value of an s-COUNTER.

A RESET operation can also be used to efficiently reuse counters between different phases of a program.

5 Alternatively, the old counters can be deleted and recreated as new counters. RESET simply resets value(c) back to zero. However, to avoid the possibility of race conditions, RESET must not be called concurrently with any other operation on the same counter. RESET ends the process, and is not intended as a
10 means of synchronization between threads.

Another thread synchronization object is a special flag, called an s-FLAG. S-FLAGS, like s-COUNTERS, have restricted allowed operations within the multithreadable code concept. S-FLAGS support SET and CHECK operations. Initially, an s-FLAG is not set.
5 A SET operation on an s-FLAG atomically sets the flag. A CHECK operation on a flag suspends until the flag is set. Once an s-FLAG is set, it remains set.

Flags and counters are provided to provide deterministic synchronization within multithreadable constructs, as previously
20 described.

In summary of the above, an s-COUNTER object has the following operations (expressed in the C programming language):

Initialize(Counter *c)

```

Finalize(Counter *c)

Increment(Counter *c, unsigned int amount)

Check(Counter *c, unsigned int value)

Reset(Counter *c)

```

5

The Initialize operation initializes the Counter object and sets its value to zero. The Finalize operation destroys the Counter object. An Increment operation increases the value of the Counter object by amount. A Check operation suspends the calling thread until the value of the Counter object is at least value. A Reset operation resets the value of the Counter object to zero.

10

15
20
25
30
35
40
45
50
55
60
65
70
75
80
85
90
95
100
105
110
115
120
125
130
135
140
145
150
155
160
165
170
175
180
185
190
195
200
205
210
215
220
225
230
235
240
245
250
255
260
265
270
275
280
285
290
295
300
305
310
315
320
325
330
335
340
345
350
355
360
365
370
375
380
385
390
395
400
405
410
415
420
425
430
435
440
445
450
455
460
465
470
475
480
485
490
495
500
505
510
515
520
525
530
535
540
545
550
555
560
565
570
575
580
585
590
595
600
605
610
615
620
625
630
635
640
645
650
655
660
665
670
675
680
685
690
695
700
705
710
715
720
725
730
735
740
745
750
755
760
765
770
775
780
785
790
795
800
805
810
815
820
825
830
835
840
845
850
855
860
865
870
875
880
885
890
895
900
905
910
915
920
925
930
935
940
945
950
955
960
965
970
975
980
985
990
995
1000

In the following simple example, a "producer thread" produces items and writes them to a buffer. A group of one or more concurrently executed "consumer threads" each independently reads the items from the buffer. The key synchronization issue is to prevent the consumer threads from reading items from the buffer that have not yet been written by the producer thread. The following program fragment gives implementations of the producer thread and consumer threads (in the C programming language) using a counter for synchronization.

20

```

Counter count;

Item buffer[NUM_ITEMS];

ProducerThread(int blockSize)

```

25

```

{
    int index = 0, c = 0;
    while (index < NUM_ITEMS) {
        buffer[index] = Produce();
        index = index + 1;
        c = c + 1;
        if (c == blockSize) {
            Increment(count, blockSize);
            c = 0;
        }
    }
}

ConsumerThread(int blockSize)
{
    int index = 0, c = blockSize;
    while (index < NUM_ITEMS) {
        if (c == blockSize) {
            Check(count, index + blockSize);
            c = 0;
        }
        Consume(buffer[index]);
        index = index + 1;
        c = c + 1;
    }
}

```

After writing a block of items to the buffer, the producer thread increments the s-COUNTER. Before reading a block of items from the buffer, a consumer thread checks the counter. If the next block of items has not yet been written to the buffer, the consumer thread suspends until enough items have been written. The program does not require that the producer and consumer threads all use the same blockSize values.

The monotonicity of counters helps guarantee deterministic synchronization and the equivalence of multithreaded and sequential execution.

If shared variables are guarded against concurrent operations, a program that uses only counter synchronization can

produce deterministic results on all executions. Moreover, if sequential execution of the program (i.e., execution ignoring the MULTITHREADED keyword) does not deadlock, multithreaded execution is guaranteed not to deadlock and to produce the same results as sequential execution. These properties are extremely useful in the testing and debugging of multithreaded programs.

Even in the absence of concurrent operations on shared variables, traditional synchronization mechanisms can introduce nondeterminacy into a program through timing dependent race conditions between threads. For example, consider the following program that uses a lock:

```
multithreaded {
  { AcquireLock(&xLock); x = x+1; ReleaseLock(&xLock); }
  { AcquireLock(&xLock); x = x*2; ReleaseLock(&xLock); }
}
```

Even though there are no concurrent operations on x, the resulting value of x is nondeterministic because of the race condition on the order in which the two threads acquire the lock. In contrast, because counters are monotonic, once a synchronization condition is enabled it remains enabled, and there is no possibility of a race condition to catch or miss a particular counter value. For example, consider the following program that uses a counter:

```
multithreaded {
  { CheckCounter(&xCount, 0); x = x+1; IncrementCounter(&xCount, 1); }
  { CheckCounter(&xCount, 1); x = x*2; IncrementCounter(&xCount, 1); }
}
```

The resulting value of x is deterministic, because the CHECKCOUNTER operations will succeed in the same order in all executions, therefore the operations on x will occur in the same order. Moreover, since sequential execution does not deadlock, multithreaded execution cannot deadlock and will always produce the same results as sequential execution.

Programs that use only counter synchronization can still be erroneously nondeterministic if they do not guard against concurrent access to shared variables. For example, consider the following program using a counter:

```
multithreaded {
    { CheckCounter(&xCount, 0); x = x+1; IncrementCounter(&xCount, 1); }
    { CheckCounter(&xCount, 0); x = x*2; IncrementCounter(&xCount, 1); }
}
```

The result of the program is nondeterministic because of the possibility of concurrent execution of the operations on x. The nondeterminacy is caused by concurrent access to a shared variable, not by a synchronization race condition.

As a simple example, consider the following program to sum the elements of a two-dimensional array:

```
void SumElements(float A[N][N], float *sum, int numThreads)
{
    int i;
    SthreadCounter counter;

    SthreadCounterInitialize(&counter);
    #pragma multithreadable mapping(blocked(numThreads))
    for (i = 0; i < N; i++) {
        int j;
        float rowSum;
        rowSum = 0.0;
        for (j = 0; j < N; j++)
```

```
        rowSum = rowSum + A[i][j];
        SthreadCounterCheck(&counter, i);
        *sum = *sum + rowSum;
        SthreadCounterIncrement(&counter, 1);
5      }
      SthreadCounterFinalize(&counter);
    }
```

Without the counter operations, multithreaded execution of
10 the for loop would not be equivalent to sequential execution,
because the iterations all modify the same *sum variable.
However, the counter operations ensure that multithreaded
execution is equivalent to sequential execution. In sequential
execution, the iterations are executed in increasing order and
15 the STHREADCOUNTERCHECK operations succeed without suspending. In
multithreaded execution, the counter operations ensure that the
operations on *sum occur atomically and in the same order as in
sequential execution. Iteration i suspends at the
20 STHREADCOUNTERCHECK operation until iteration i - 1 has executed the
STHREADCOUNTERINCREMENT operation.

Conditions are carved out to prevent concurrent access to
shared variables using counters. Essentially, each pair of
operations on a shared variable must be separated by a
transitive chain of counter operations. If these conditions can
25 be shown to hold in any one execution of the program, they must
hold in all executions of the program. Therefore, if sequential
execution satisfies the conditions, multithreaded execution is
also guaranteed to satisfy the conditions, hence produce the

same results as sequential execution. This result forms the basis of a powerful methodology for developing multithreaded programs using sequential reasoning, testing, and debugging techniques.

5 All the programs using counters so far discussed satisfy the conditions on shared variables, therefore are guaranteed to be deterministic. In addition, the program examples described herein have equivalent multithreaded and sequential execution. The cost of increased determinacy is decreased concurrency. Synchronization using counters provides an effective means of
10 controlling this tradeoff between determinacy and concurrency.

Counters can also be used as a stronger form of lock synchronization, providing sequential ordering in addition to mutual exclusion on a critical section. With the traditional
15 implementation of mutual exclusion using a pair of lock operations, the order in which threads enter the critical section is nondeterministic. This is desirable in terms of maximizing concurrency, but is undesirable in terms of reasoning, testing, and debugging, and simply might not satisfy
20 the desired program specification. Replacing the pair of lock operations with a pair of counter operations can guarantee deterministic results, at the cost of decreased opportunities for concurrency.

Consider the computation of a result object formed by accumulating a series of independent subresults that are computed concurrently. For example, the result object could be a linked list and the accumulate operation could be an append, or the result object could be a summation and the accumulate operation could be an addition. Mutual exclusion is required to prevent interference between multiple concurrent accumulate operations on the result object.

The following program implements the computation with one thread computing each subresult, and a pair of lock operations to provide mutual exclusion:

```
CompositeItem result;
Lock resultLock;

InitializeLock(&resultLock);
multithreaded for (i = 0; i < N; i++) {
    SingleItem subresult = compute(i);
    AcquireLock(&resultLock);
    accumulate(&result, subresult);
    ReleaseLock(&resultLock);
}
FinalizeLock(&resultLock);
```

Only one thread can hold RESULTLOCK at any given time, thereby ensuring mutual exclusion of the ACCUMULATE operations. However, if the accumulate operation is not associative and determinacy of results is desired, some other mechanism is required to ensure sequential (or at least deterministic) ordering, in addition to mutual exclusion. For example, neither appending an item to a linked list, nor floating point addition are associative operations. With both these examples, the

above program may produce different results on repeated executions.

The following program implements the computation with the pair of lock operations replaced with a pair of counter operations, to provide both mutual exclusion and sequential ordering:

```

CompositeItem result;
Counter resultCount;
10 InitializeCounter(&resultCount);
   multithreaded for (i = 0; i < N; i++) {
       SingleItem subresult = compute(i);
       CheckCounter(&resultCount, i);
       accumulate(&result, subresult);
       IncrementCounter(&resultCount, 1);
15   }
   FinalizeCounter(&resultCount);

```

As with the lock program, only one ACCUMULATE operation can execute at any given time. However, the accumulate operations are now additionally constrained to execute in sequential order. RESULTCOUNT[i] = i indicates that thread i-1 has completed its accumulate operation. The counter program has greater determinacy at the cost of less concurrency. With the lock

25 program, an ACCUMULATE operation can execute concurrently with compute operations in all other threads. With the counter program, an ACCUMULATE operation can only execute concurrently with compute operations in higher numbered threads.

The optimal tradeoff between determinacy and concurrency

30 has to be made on a case by case basis. Counters are a powerful mechanism for providing sequential ordering on top of mutual

exclusion in the many cases where determinacy is important and the performance consequences of less concurrency are not great.

The Sthreads Interface

5 The code produced according to the present application can be expressed using the Multithreaded C pragma notation. As described previously, there is a direct correspondence between the pragma notation for thread creation and the Sthreads library functions that support thread creation. As a simple example, the following is a program implemented using Sthreads:

```

10      typedef struct {
15          float (*A) [N];
          float *sum;
          SthreadCounter *counter;
        } LoopArgs;

        void LoopBody(int i, int notused1, int notused2, LoopArgs *args)
        {
20            int j;
            float rowSum;
            rowSum = 0.0;
            for (j = 0; j < N; j++)
                rowSum = rowSum + (args->A) [i] [j];
            SthreadCounterCheck(args->counter, i);
25            *(args->sum) = *(args->sum) + rowSum;
            SthreadCounterIncrement(args->counter, 1);
        }

        void SumElements(float A[N] [N], float *sum, int numThreads)
30        {
            int i;
            SthreadCounter counter;
            LoopArgs args;

35            SthreadCounterInitialize(&counter);
            args.A = A;
            args.sum = sum;
            args.counter = &counter;
            SthreadRegularForLoop(
40                (void (*)(int, int, int, void *)) LoopBody,
                (void *) &LoopArgs,
                0, STHREAD_CONDITION_LT, N, 1,

```

```
1, STHREAD_MAPPING_BLOCKED, numThreads,  
    STHREAD_PRIORITY_PARENT, STHREAD_STACK_SIZE_DEFAULT);  
    SthreadCounterFinalize(&counter);  
}
```

5

Although this program is syntactically more complicated than the Multithreaded C version, it is considerably less complicated than the same program expressed using Windows NT threads. The mechanics of creating threads, assigning iterations to threads, and waiting for thread termination is handled within the Sthreads library call.

The Sthreads multithreaded programming system is implemented as a transparent add-on to an existing program development system, e.g., a compiler or interpreter, or other program development environment. The notation and implementation allows multithreaded and multithreadable code constructs to be directly translated into a high-level structured thread library. This translation is implemented as a preprocessor that can be transparently called prior to the standard compilation phase in an existing program development system.

For example, when integrated with the Microsoft Visual C++ programming system, the standard CL (Compiler-Linker) is replaced by a special Sthreads tool that calls the Sthreads preprocessor on program files, then calls the standard (renamed) Visual C++ CL.

Integration of Sthreads with existing programming systems allows programmers new flexibility without adopting new programming systems to use the power of multithreading. They can use their standard editor, debugger, compiler, etc., and simply
 5 add Sthreads to the system. It also means that Sthreads piggybacks on the quality of code generation and error analysis of the underlying development system.

Preprocessing had previously been used for many kinds of program "source-to-source" transformations. Sthreads in
 10 contrast, implements a full-fledged, sophisticated multithreaded programming system by using a preprocessing integrated with a standard program development environment.

One implementation has been created in the ANSI C language, thereby defining a "Multithreaded C" language. A structured
 15 thread library (Sthreads) is called by the languages. In both Multithreaded C and Sthreads, thread creation constructs are multithreaded variants of sequential "block" (i.e., sections of program code distinctly defined by conventional program constructs) and "for loop" constructs. In the Multithreaded C
 20 implementation, these constructs are supported as pragma annotations to a sequential program. With Sthreads, exactly the same constructs are supported as library calls. At both levels, synchronization objects and operations are supported as Sthreads library calls.

In this embodiment, the Sthreads library for Windows NT can be implemented as a very thin layer on top of the Win32 thread API. As a consequence, there is essentially no performance overhead associated with using Sthreads or Multithreadable C, as compared
5 to using Win32 threads directly.

Multithreaded C is implemented as a portable source-to-source preprocessor that directly transforms annotated blocks and for loops into equivalent calls to the Sthreads library. The programmer has the option of either using the pragma annotations and preprocessor or making Sthreads calls directly. The Sthreads
10 library and Multithreaded C preprocessor are integrated with Microsoft Developer Studio Visual C++. Building a project preferably automatically invokes the preprocessor where necessary and links with the Sthreads library.

Multithreadable blocks and for loops are implemented as a
15 sequence of CREATETHREAD calls followed by a WAITFORSINGLEOBJECT call on an event. Terminating threads perform an INTERLOCKEDDECREMENT call on a counter, and the last thread to terminate performs a SETEVENT call on the event. Flags are implemented directly as
20 Win32 events. Counters are implemented as linked lists of Win32 events, with one event for every value on which some thread is waiting. Locks are implemented directly as Win32 critical sections. Barriers are implemented as a pair of Win32 events and a Win32 critical section.

An important issue of the multithreading operation comes about when considering multiple processors. The hardware and operating systems of modern technology allow for multiprocessor systems. Current operation in multiprocessor systems, however, have often simply operated on one but not the other processor. By multithreading in this way, the different threads can actually be executed on the different processors.

In operation, when a multithreading indicator (such as a "compile as multithreaded" flag/button/environment-variable) is set, both multithreadable and multithreaded blocks/loops are compiled to multithreaded code. When the multithreading indicator is not set, the multithreaded blocks/loops are compiled to multithreaded code and the multithreadable blocks/loops are compiled into ordinary sequential code. This allows a programmer to mix constructs that only have a multithreaded meaning (e.g., real-time control and systems programming uses of threads) with constructs that can be compiled into threads for multiprocessor performance or compiled into equivalent sequential code when developing and debugging.

The invention allows a program to run as fast as a sequential program on one processor, but significantly faster on multiprocessors, without recompilation, relinking, or reconfiguration. The invention thus allows a program to adapt dynamically to changing resources. Use of monotonic flags and

monotonic counters makes embodiments of the invention reliable and timely.

The mapping of Statements/Iterations onto threads is relatively simple. One thread is used for each statement/chunk, or for a small number of statements/chunks.

A Typical for loop may have thousands or millions of iterations. The Overhead associated with assigning units of work to threads is significant. The present application defines assigning the iterations in contiguous "chunks". Significant unit of work should be performed by each chunk.

For example:

```
#PRAGMA MULTITHREADABLE CHUNKSIZE(1000), MAPPING(BLOCKED(T))

FOR (I = 0; I < N; I++)

    A[I] = B[I];
```

Interaction of Chunksize and Mapping is described in the following example:

```
#PRAGMA MULTITHREADABLE CHUNKSIZE(2), MAPPING(BLOCKED(4))

FOR (I = 50; I >= 10; I = I - 2)

    DoSOMETHING(I);
```

The Complete Sthreads Library includes a number of statements:

Processor Management: STHREADSGETNUMPROCESSORSPRESENT,
STTHREADSSETNUMPROCESSORSUSED, STHREADSGETPROCESSORSPRESENT,
STTHREADSSETPROCESSORSUSED.

Thread Creation: STHREADSBLOCK, STHREADSREGULARFORLOOP.

5 Thread Scheduling: STHREADSGETCURRENTPRIORITY,
STTHREADSSETCURRENTPRIORITY.

Flags: STHREADSFLAGINITIALIZE, STHREADSFLAGFINALIZE, STHREADSFLAGSET,
STTHREADSFLAGCHECK, STHREADSFLAGRESET.

Counters: STHREADSCOUNTERINITIALIZE, STHREADSCOUNTERFINALIZE,
10 STHREADSCOUNTERINCREMENT, STHREADSCOUNTERCHECK, STHREADSCOUNTERRESET.

Locks: STHREADSLOCKINITIALIZE, STHREADSLOCKFINALIZE,
STTHREADSLOCKACQUIRE, STHREADSLOCKRELEASE.

Barriers: STHREADSBARRIERINITIALIZE, STHREADSBARRIERFINALIZE,
STTHREADSBARRIERPASS, STHREADSBARRIERRESET.

15 Examples of computer program code implementing each of
these constructs are set forth in the appendix.

FIG. 4 is a process flowchart showing a method for
compiling multithreadable code in accordance with one embodiment
of the invention. The computer program source code text 400
20 includes annotations defining multithreadable code constructs
(and, optionally, multithreaded code constructs) and any
necessary processor management, thread creation, and
synchronization constructs (such as monotonic flags and
counters). If a multithreading indicator is set 401, the source

code text 400 is processed by a pre-processor 402 that parses the source into an expanded computer program text. The expanded computer program text includes inserted calls to an Sthreads library 406 to invoke multithreaded program operations wherever
 5 a source code annotation called for multithreadable functionality. A conventional compiler 406 then communicates with a linker 410 which links pre-existing routines from the Sthreads library 406 with the output of the compiler to create an executable module 412.

10 If the multithreading indicator is not set 401, the original computer program source code text 400 is compiled and linked in conventional fashion, with each section of multithreadable code constructs compiled as sequentially executing code. Annotations not recognized by the compiler 408
 15 are ignored.

A convenient implementation shortcut that permits ready use of conventional compilers and linkers is to rename a pre-existing compiler-linker executable file to a new name, and assign the old name of the compiler-linker executable file to
 20 the pre-processor. The pre-processor then can call the compiler-linker executable file when needed by invoking the new name.

Synchronization Using Locks

Locks are provided to express nondeterministic synchronization, usually mutual exclusion, within multithreaded BLOCKS and FOR loops. Sthread locks support the usual ACQUIRE and
 5 RELEASE operations. The order in which concurrent ACQUIRE operations succeed is nondeterministic. Therefore, there is very little use for locks within multithreadable blocks and FOR loops. As a simple example, consider the following program to sum the elements of a two-dimensional array:

```

10 void SumElements(float A[N][N], float *sum, int numThreads)
   {
       int i;
       SthreadLock lock;

15     SthreadLockInitialize(&lock);
       #pragma multithreaded mapping(blocked(numThreads))
       for (i = 0; i < N; i++) {
           int j;
           float rowSum;
           rowSum = 0.0;
20           for (j = 0; j < N; j++)
               rowSum = rowSum + A[i][j];
           SthreadLockAcquire(&lock);
           *sum = *sum + rowSum;
           SthreadLockRelease(&lock);
25       }
       SthreadLockFinalize(&lock);
   }

```

30 Like the flag operations in the program, the lock operations in this program ensure that the operations on *SUM occur atomically. However, unlike the flag operations, the lock operations do not ensure that the operations on *SUM occur in the same order as in sequential execution, or even in the same order

each time the program is executed. Therefore, since floating-point addition is not associative, the program may produce different results each time it is executed. However, because execution order is less restricted, this program allows more concurrency than the program described above. This is an example of the commonly-occurring tradeoff between determinacy and efficiency.

Synchronization Using Barriers

S-thread barriers are provided to express collective synchronization of a group of threads in cases when thread termination and recreation is too expensive. The barriers described herein support the usual PASS operation. All the threads in a group must enter the PASS operation before all the threads in the group are allowed to leave the Pass operation. In current systems, the cost of N threads executing a PASS operation is less than the cost of creating and terminating N threads. Therefore, a typical use of barriers is to replace a sequence of multithreadable loops with a single multithreaded loop containing a sequence of barrier PASS operations. However, with modern lightweight thread systems such as Windows NT, we are discovering that barriers are required for efficiency in very few circumstances.

A number of examples are described herein.

Trivial Example: Independent Iterations

```

INT ARRAYSUM(FLOAT A[N] [N] )
/* SUMS THE ELEMENTS OF A 2-DIMENSIONAL ARRAY. */
{
5   FLOAT SUM, ROWSUM[N] ;
   INT I;

   SUM = 0.0;
   #PRAGMA MULTITHREADABLE
10  FOR (I = 0; I < N; I++) {
       INT J;
       ROWSUM[I] = 0.0;
       FOR (J = 0; J < N; J++)
           ROWSUM[I] = ROWSUM[I] + A[I] [J];
15  }
   FOR (I = 0; I < N; I++)
       SUM = SUM + ROWSUM[I];
   RETURN SUM;
}

```

A more difficult example is shown in the following.

```

INCORRECT EXAMPLE: NONDETERMINACY
INT ARRAYSUM(FLOAT A[N] [N] )
/* SUMS THE ELEMENTS OF A 2-DIMENSIONAL ARRAY. */
{
25  FLOAT SUM;
   INT I;
   STHREADSLOCK SUMLOCK;

30  STHREADSLOCKINITIALIZE(&SUMLOCK);
   SUM = 0.0;
   #PRAGMA MULTITHREADABLE
   FOR (I = 0; I < N; I++) {
35     INT J;
       FLOAT ROWSUM = 0.0;
       FOR (J = 0; J < N; J++)
           ROWSUM = ROWSUM + A[I] [J];
       STHREADSLOCKACQUIRE(&SUMLOCK);
40     SUM = SUM + ROWSUM;
       STHREADSLOCKRELEASE(&SUMLOCK);
   }
   STHREADSLOCKFINALIZE(&SUMLOCK);
}

```

```

    RETURN SUM;
}
INT ARRAYSUM(FLOAT A[N] [N] )
/* SUMS THE ELEMENTS OF A 2-DIMENSIONAL ARRAY. */
5 {
    FLOAT SUM;
    INT I;
    STHREADSCOUNTER SUMCOUNT;

10    STHREADSCOUNTERINITIALIZE(&SUMCOUNT);
    SUM = 0.0;
    #PRAGMA MULTITHREADABLE
    FOR (I = 0; I < N; I++) {
        INT J;
15        FLOAT ROWSUM = 0.0;
        FOR (J = 0; J < N; J++)
            ROWSUM = ROWSUM + A[I] [J];
        STHREADSCOUNTERCHECK(&SUMCOUNT, I);
        SUM = SUM + ROWSUM;
20        STHREADSCOUNTERINCREMENT(&SUMCOUNT, 1);
    }
    STHREADSCOUNTERFINALIZE(&SUMCOUNT);
    RETURN SUM;

```

25 As can be seen, iterations cannot be executed as separate threads because of nondeterminacy in the top. However, the counters allow determinacy between the system therefore enabling the system to be multithreaded.

30 Single-Writer Multiple-Reader Broadcast

Counters can be used to provide elegant, flexible, and efficient dataflow synchronization between a single writer and multiple readers of a sequence of items written to an array. In this synchronization pattern, reading an item does not remove it 35 from the sequence—each reader independently reads the entire shared array. Because a counter has multiple thread suspension

queues, a single counter object can be used to synchronize the writer thread and any number of completely independent reader threads, with each thread potentially having a different granularity of synchronization. The writer thread incrementing the counter broadcasts the availability of data to the entire set of reader threads.

The following program demonstrates the single-write multiple-reader broadcast pattern with synchronization on every item:

```

10 void Writer(Item *data, int n, Counter *dataCount)
   {
       int i;
       for (i = 0; i < n; i++) {
           data[i] = GenerateItem(i);
           IncrementCounter(dataCount, 1);
       }
   }

20 void Reader(Item *data, int n, Counter *dataCount)
   {
       int i;
       for (i = 0; i < n; i++) {
           CheckCounter(dataCount, i+1);
           UseItem(data[i]);
       }
   }

25 ...
   Item data[N];
   Counter dataCount;
   int r;

30 InitializeCounter(&dataCount);
   multithreaded {
       Writer(data, N, dataCount);
       multithreaded for (r = 0; r < numReaders; r++)
35         Reader(data, N, dataCount);
   }
   FinalizeCounter(&dataCount);
   ...

```

One WRITER thread and an arbitrary number of Reader threads are executed concurrently, with communication through the shared data array, and synchronization through the DATACOUNT counter. At

any point, some Reader threads may be suspended in their
CHECKCOUNTER operation, waiting for the Writer thread to increment
DATACOUNT, while other Reader threads may be reading data items
that have previously been written. The READER threads execute
5 independently of each other and do not synchronize their actions
in any manner. The synchronization pattern is strictly a one-to-
many broadcast from the WRITER thread to the READER threads.

Synchronization on every item that is written and read may
be too expensive if the time taken to generate and use an item
10 is too small. The single-reader multiple-writer broadcast
pattern can be generalized to allow the writer and each reader
thread to synchronize on a block of items instead of on
individual items. The following program adds an individual
granularity of blocked synchronization to the writer and each
15 reader thread:

```
void Writer(Item *data, int n, Counter dataCount, int blockSize)
{
    int i;
    for (i = 0; i < n; i++) {
        data[i] = GenerateItem(i);
        if ((i+1)%blockSize == 0)
            IncrementCounter(dataCount, blockSize);
    }
    IncrementCounter(dataCount, n-(n/blockSize)blockSize);
}

20 void Reader(Item *data, int n, Counter *dataCount, int blockSize)
{
    int i;
    for (i = 0; i < n; i++) {
        if (i%blockSize == 0)
            CheckCounter(dataCount, min(i+blockSize, n));
        UseItem(data[i]);
    }
}

30
```

The WRITER and READER threads now increment and check the
DATACOUNT counter in multiples of BLOCKSIZE and write and read the
data array in blocks of items. There is no requirement that
BLOCKSIZE be the same in all threads. Different threads can be
5 passed different BLOCKSIZE based on their individual performance
characteristics and requirements. This pattern is now extremely
flexible and easily adaptable with regard to practical
performance tuning.

The single-writer multiple-reader broadcast pattern is a
10 dataflow synchronization pattern that occurs in many diverse
applications of threads to multiprocessing. For example, in the
Paraffins Problem, an array of molecules of a certain size can
be generated by one thread and concurrently read by other
threads that in turn generate arrays of larger molecules. The
15 pattern is very different from, for instance, the multiple-
writers multiple-readers bounded-buffer problem, which is
elegantly solved using semaphores. Just as counters are not well
suited to implementing bounded buffers, semaphores and other
traditional synchronization mechanisms are not well suited to
20 implementing the single-writer multiple-reader broadcast
pattern.

Another Example Application: Aircraft Route Optimization

The Aircraft Route Optimization Problem is part of the U.S. Air Force Rome Laboratory C3I Parallel Benchmark Suite. For this application, we achieved better performance using Sthreads on a quad-processor Pentium Pro system running Windows NT than the best reported results for message-passing programs running on expensive Cray and SGI supercomputers with up to 64 processors. The flexibility of shared-memory, lightweight multithreading, and sequential development methods allowed us to develop a much more sophisticated and efficient algorithm than would be possible on a message-passing supercomputer.

The C3I Parallel Benchmark Suite

The U.S. Air Force Rome Laboratory C3I Parallel Benchmark Suite consists of eight problems chosen to represent the essential elements of real C3I (Command, Control, Communication, and Intelligence) applications. Each problem consists of the following:

A problem description giving the inputs and required outputs.

An efficient sequential program (written in C) to solve the problem.

The benchmark input data.

A correctness test for the benchmark output data.

For some of the problems, a parallel message-passing program is also provided. Rome Laboratory maintains a publicly accessible database of reported performance results.

The C3I Parallel Benchmark Suite provides a good framework
5 for evaluating our structured multithreaded programming system. The problems are computationally intensive and involve a variety of complex algorithms and data structures. The sequential program provides us with a good starting point and a fair basis for performance comparison. The performance database allows us
10 to compare our results with those of other researchers. For these reasons, we are developing multithreaded solutions to several of the C3I Parallel Benchmark Suite problems.

The task in the Aircraft Route Optimization Problem is to find the lowest-risk path for an aircraft from an origin point
15 to a set of destination points in the airspace over an uneven terrain. The risk associated with each transition in the airspace is determined by its proximity to a set of threats. The problem involves realistic constraints on aircraft speed and maneuverability. The aircraft is also constrained to fly above
20 the underlying terrain and beneath a given ceiling altitude.

The problem is essentially the single-source, multiple-destination shortest path problem with a large, sparsely connected graph. The airspace for the benchmark is 100 km by 100 km in area and 10 km in altitude, discretized at 1 km intervals.

The 100,000 positions in space correspond to 2,600,000 nodes in the graph, since each position can be reached from 26 different directions. Because of aircraft speed and maneuverability constraints, each node is connected to only nine or ten geographically adjacent nodes. Therefore, the graph consists of approximately 2.6 million nodes and 26 million edges.

The sequential algorithm to solve the Aircraft Route Optimization Problem is based on a queue of nodes. Initially the queue is empty except for the origin node. At each step, one node is removed from the queue. Valid transitions from this source node to all adjacent destination nodes are considered. For each destination node, if the path to the node via the source node is shorter than the current shortest path to the node, the path to the node is updated and the node added to the queue. The algorithm continues until the queue is empty, at which stage the shortest paths to all reachable nodes have been computed.

The queue is ordered on path length so that shorter paths are expanded before longer paths. This has a significant effect on performance. Without ordering, longer paths are expanded, then discarded when shorter paths to the same points are expanded later in the computation. However, whether the queue is ordered, partially ordered, or unordered does not affect the results of the algorithm.

The most straightforward approach to obtaining parallelism in the Aircraft Route Optimization Problem is to geographically partition the airspace into blocks, with one thread (or process) responsible for each block. Each thread runs the sequential
5 algorithm on its own block using its own local queue and periodically exchanges boundary values with neighboring blocks. This approach is particularly appealing on distributed-memory, message-passing platforms, because memory can be permanently distributed according to the blocking pattern. If the threads
10 execute a reasonably large number of iterations between boundary exchanges, good load balance can be achieved.

The problem with this algorithm is that, as the number of blocks/threads is increased the total amount of computation also increases. Therefore, any speedup is based on an increasingly
5 inefficient underlying algorithm. At any time, the local queues in most blocks contain paths that are too long and are irrelevant to the actual shortest paths. The processors are kept busy performing computation that is later discarded. At any
given time, it is only productive to work on an irregular and
20 unpredictable subset of the graph. However, irregular and adaptive blocking schemes do not solve the problem, since there is usually equal work available in all blocks. The issue is the distinction between productive and unproductive work.

Our solution is to statically partition the airspace into a large number of blocks and to use a much smaller number of threads. A measure of the average path length is maintained with each local queue. At each step, the blocks with local queues
5 containing the shortest paths are assigned to the threads. Therefore, the subset of blocks that are active and the assignment of blocks to threads change dynamically throughout program execution. This algorithm takes advantage of the symmetric multiprocessing model, in which all threads can access
10 the entire memory space with uniform cost. It also takes advantage of the lightweight multithreading model to achieve good load balance, since the workload within each thread at each step is highly variable.

The ability to develop, test, and debug using sequential
5 methods was crucial in the development of this sophisticated multithreaded algorithm. The entire program was tested and debugged in sequential mode before multithreaded execution was attempted. In particular, development of the complex boundary exchange and queue update algorithms would have been
20 considerably more difficult in multithreaded mode.

The ability to analyze and tune performance using sequential methods was also very important. Good performance depended on exposing enough parallelism without significantly increasing the total amount of computation. We determined

efficient values for the number of blocks, the number of threads, and the number of iterations between boundary exchanges by measuring computation times and operation counts of the multithreaded program in running in sequential mode. This detailed analysis would have been very difficult to perform in multithreaded mode. We avoided memory contention in multithreaded mode by avoiding cache misses in sequential mode. The analysis of memory access patterns in sequential mode is much simpler than in multithreaded mode.

All Pairs Shortest Paths Example

This example describes the algorithmic and performance advantages of counter synchronization. In the example, a counter is used as a less restrictive, and consequently more efficient, replacement for a barrier. The example program is a multithreaded solution to the all-pairs shortest-path problem using the Floyd-Warshall algorithm. Using traditional synchronization mechanisms, this problem can be solved using one barrier or, more efficiently, an array of condition variables. We show how the efficient solution can be implemented using a single counter instead of an array of condition variables. We give timing measurements comparing the performance of the barrier, condition variable, and counter algorithms.

The all-pairs shortest-path problem takes as input the edge-weight matrix of a weighted directed graph, and returns the matrix of shortest-length paths between all pairs of vertices in the graph. The graph is required to have no cycles of negative length, and the weight of the edge from a vertex to itself is required to be zero.

The following program solves the all-pairs shortest-path problem using the sequential Floyd-Warshall algorithm:

```

10  VOID SHORTESTPATHS1 (INT EDGE[N] [N], INT PATH[N] [N])
    {
        INT K, I, J;
        PATH[0..N-1] [0..N-1] = EDGE[0..N-1] [0..N-1];
        FOR (K = 0; K < N; K++)
15     FOR (I = 0; I < N; I++)
        FOR (J = 0; J < N; J++) {
            INT NEWPATH = PATH[I] [K] + PATH[K] [J];
            IF (NEWPATH < PATH[I] [J]) PATH[I] [J] = NEWPATH;
20     }
    }

```

Initially, $PATH[I][J]$ is assigned $EDGE[I][J]$, for all i and j . (For brevity, we use a notational shorthand for array assignment.) After the k th iteration, $PATH[I][J]$ is the shortest path from vertex i to vertex j with intermediate vertices only in vertices 0 to k . Therefore, after N iterations, $PATH[I][J]$ is the shortest path from vertex i to vertex j with no restrictions on the intermediate vertices.

The following program solves the all-pairs shortest path problem using a multithreaded version of the Floyd-Warshall algorithm, with a barrier for thread synchronization:

```

void ShortestPaths2(int edge[N][N], int path[N][N], int numThreads)
{
    int t;
    Barrier b;
5
    path[0..N-1][0..N-1] = edge[0..N-1][0..N-1];
    InitializeBarrier(&b, numThreads);
    multithreaded for (t = 0; t < numThreads; t++) {
        int k, i, j;
10
        for (k = 0; k < N; k++) {
            for (i = t*N/numThreads; i < (t+1)*N/numThreads; i++)
                for (j = 0; j < N; j++) {
                    int newPath = path[i][k] + path[k][j];
                    if (newPath < path[i][j]) path[i][j] = newPath;
15
                }
            PassBarrier(&b);
        }
    }
    FinalizeBarrier(&b);
20
}

```

The multithreaded outer loop creates NUMTHREADS threads. Each thread executes the N iterations of the Floyd-Warshall algorithm on a subset of the rows of the path matrix. To keep the iterations synchronized, the threads pass through an N-way barrier at the end of each iteration. There are no sharing violations on the concurrent accesses to PATH across the threads, because the algorithm will never assign to PATH[I][K] or PATH[K][J] during iteration k.

The barrier algorithm successfully divides the work among an arbitrary number of threads. However, in requiring that all threads complete each iteration before any thread begins the next iteration, the algorithm does not express the full opportunities for concurrency inherent in the data dependencies. As a consequence, the program is less than optimally efficient. N-way synchronization at the barrier is a bottleneck that creates delays on entry and exit, and processor load imbalance

can occur if all threads do not reach the barrier simultaneously.

A More Efficient Multithreaded Solution Using Condition Variable Synchronization

The following program solves the all-pairs shortest path problem using a more efficient multithreaded version of the Floyd-Warshall algorithm, with an array of N condition variables for thread synchronization:

```

10 void ShortestPaths3(int edge[N][N], int path[N][N], int numThreads)
    {
        int k, t;
        Condition kDone[N];
        int kRow[N][N];

15     path[0..N-1][0..N-1] = edge[0..N-1][0..N-1];
        for (k = 0; k < N; k++) InitializeCondition(&kDone[k]);
        kRow[0] = path[0][0..N-1];
        SetCondition(&kDone[0]);
        multithreaded for (t = 0; t < numThreads; t++) {
20             int k, i, j;
            for (k = 0; k < N; k++) {
                CheckCondition(&kDone[k]);
                for (i = t*N/numThreads; i < (t+1)*N/numThreads; i++) {
25                     for (j = 0; j < N; j++) {
                        int newPath = path[i][k] + kRow[k][j];
                        if (newPath < path[i][j]) path[i][j] = newPath;
                    }
                    if (i == k+1) {
30                         kRow[k+1][0..N-1] = path[k+1][0..N-1];
                        SetCondition(&kDone[k+1]);
                    }
                }
            }
        }
35     for (k = 0; k < N; k++) FinalizeCondition(&kDone[k]);
    }

```

As with the barrier algorithm, each thread executes the N iterations of the Floyd-Warshall algorithm on a subset of the rows of the PATH matrix. However, each thread can individually continue with its next iteration as soon as the necessary data is available, instead of waiting for the previous iteration to

complete in all the other threads. Condition variable `KDONE[k]` is set when row `k` of the `PATH` matrix has been computed in iteration `k-1`. Each thread waits on `KDONE[k]` before executing iteration `k`. To avoid sharing violations, row `k` of the `PATH` matrix computed in iteration `k-1` is stored in `KRow[k]`.

The condition variable algorithm avoids the inefficiencies associated with barrier synchronization. Threads synchronize individually, rather than in an `N`-way bottleneck, and faster threads can execute many iterations ahead of slower threads. Potentially, the `N` threads can be executing in up to `N` different iterations. One extra cost of this algorithm is the storage for the `KRow` matrix. However, the most significant extra cost is allocation of `N` condition variables.

The following program solves the all-pairs shortest path problem using the efficient multithreaded version of the Floyd-Warshall algorithm, with a single counter for thread synchronization in place of `N` condition variables:

```

void ShortestPaths3(int edge[N][N], int path[N][N], int numThreads)
{
    int k, t;
    Counter kCount;
    int kRow[N][N];

    path[0..N-1][0..N-1] = edge[0..N-1][0..N-1];
    InitializeCounter(&kCount);
    kRow[0] = path[0][0..N-1];
    multithreaded for (t = 0; t < numThreads; t++) {
        int k, i, j;
        for (k = 0; k < N; k++) {
            CheckCounter(&kCount, k);
            for (i = t*N/numThreads; i < (t+1)*N/numThreads; i++) {
                for (j = 0; j < N; j++) {
                    int newPath = path[i][k] + kRow[k][j];
                    if (newPath < path[i][j]) path[i][j] = newPath;
                }
                if (i == k+1) {
                    kRow[k+1][0..N-1] = path[k+1][0..N-1];
                    IncrementCounter(&kCount, 1);
                }
            }
        }
    }
    FinalizeCounter(&kCount);
}

```

Operations on N different values of the single counter replace operations on N different elements of the array of condition variables. The algorithm has the same performance advantages over the barrier algorithm, without the cost of statically allocating and maintaining N synchronization objects. Internally, the counter may create synchronization objects for the distinct counter values on which threads are suspended. However, in practice, the number of these objects in existence at any given time is likely to be a small fraction of N.

Three Synchronization Patterns Example

Three examples of practical synchronization patterns are described that can be expressed more elegantly (and often more

efficiently) using counters than with traditional synchronization mechanisms. For each of these synchronization patterns, a small example program is provided to demonstrate the pattern and a description of the importance of the pattern to real problems. This is far from an exhaustive list of patterns to which counters can usefully be applied. Counters are equally applicable to many other situations, particularly dataflow style synchronization patterns arising in the application of threads to multiprocessing.

Counters can often be used to replace traditional barrier synchronization with a less restrictive form of "ragged" barrier. With a ragged barrier, each thread waits at the barrier point only until its own individual data dependencies have been satisfied, instead of until the data dependencies of all threads have been satisfied, as with a traditional barrier. We have already given one example of this pattern in Section 0, with the multithreaded Floyd-Warshall algorithm to solve the all-pairs shortest-path problem. In this section, we give another more straightforward example, based on boundary exchange in a time-stepped simulation.

Consider a time-stepped simulation of a one-dimensional object subdivided into N cells. The state of internal cell i at time t is a function of the states of cells $i-1$, i , and $i+1$ at time $t-1$. The states of the leftmost and rightmost cells remain

constant over time. An example is simulation of heat transfer along a metal rod. Similar boundary exchange requirements occur in most multithreaded simulations of physical systems in one or more dimensions. These requirements are traditionally satisfied
 5 using barrier synchronization.

The following program implements the simulation using one thread for each cell, with traditional barrier synchronization between threads before cell state exchanges and updates at each time step:

```

10 float state[N];
    Barrier b;
    ...
    state[0..N-1] = initial cell states;
    InitializeBarrier(&b, N-2):
15 multithreaded for(i = 1; i < N-1; i++) {
    float leftState, rightState;
    for (t = 1; t <= numSteps; t++) {
        PassBarrier(&b);
        leftState = state[i-1];
        rightState = state[i+1];
20 PassBarrier(&b);
        state[i] = f(leftState, state[i], rightState);
    }
25 FinalizeBarrier(&b):

```

All threads synchronize at the barrier twice every time step: once before exchanging cell states, and again before updating cell states. However, complete barrier synchronization
 30 between all threads is unnecessarily restrictive. The conditions for safely exchanging and updating the cell states involve dependencies between pairs of neighboring cells, not across all cells. As a consequence of using barriers, the performance of

the program is potentially subject to synchronization bottleneck and load imbalance problems.

The following program implements the same simulation using an array of counters to provide ragged barrier synchronization between threads:

```

float state[N];
Counter c[N];
...
state[0..N-1] = initial cell states;
for (i = 0; i < N; i++) CounterInitialise(&c[i]);
IncrementCounter(&c[0], 2*numSteps);
IncrementCounter(&c[N-1], 2*numSteps);
multithreaded for (i = 1; i < N-1; i++) {
    float leftState, rightState, myState = state[i];
    for (t = 1; t <= numSteps; t++) {
        CheckCounter(&c[i-1], 2*t-2); leftState = state[i-1];
        CheckCounter(&c[i+1], 2*t-2); rightState = state[i+1];
        IncrementCounter(&c[i], 1);
        myState = f(leftState, myState, rightState);
        CheckCounter(&c[i-1], 2*t-1);
        CheckCounter(&c[i+1], 2*t-1);
        state[i] = myState;
        IncrementCounter(&c[i], 1);
    }
}
for (i = 0; i < N; i++) FinalizeCounter(&c[i]);

```

As with the traditional barrier algorithm, the threads synchronize every time step before exchanging cell states, and again before updating cell states. However, the synchronization is between pairs of neighboring threads via an array of counters. $c[i] = 2 \cdot T - 1$ indicates that thread i has finished reading both neighboring cell states in time step T , and $c[i] = 2 \cdot T$ indicates that thread i has completed time step T . Pairwise synchronization removes the synchronization bottleneck of a traditional barrier and reduces load imbalance by allowing some threads to execute ahead of other threads. The barrier could be

made even more ragged using separate counters to synchronize with left and right neighbors.

The major cost in the implementation of ragged barriers using counters is the need for N counter objects instead of one barrier object. However, the number of counters needed is proportional to the number of threads, not to the problem size. This cost is unlikely to be a practical problem on modern computer systems.

The present application can be used in multithreaded programming system, with any single or multiprocessor computers. Example multithreaded programming systems include Windows NT, UNIX/Pthreads and Java.

Other examples than those discussed above can of course be used. While the three examples discussed above are computationally intensive, other computationally intensive systems include volume rendering, terrain masking, threat analysis, protein folding, and molecular dynamics simulation.

As can be seen from the above, the system of the present application is highly advantageous and produces significant advantages.

Although only a few embodiments have been disclosed in detail above, other modifications are possible, and would be understood by those having ordinary skill in the art reading the application. For example, although this application has only

described certain operating systems which capable of handling multiple threads, it should be understood that other operating systems could be provided. A non-exhaustive list of operating systems includes Windows NT, Windows 2000, Java, UNIX, Linux or
5 any other type system.

All such modifications are intended to be encompassed within the following claims, in which:

1

2

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100

What is claimed is:

1. A method of synchronizing threads in a multiple thread system, comprising:

defining an entity which maintains a count of values which increases the value maintained by the object; and

defining a check operation for said element in which, during the checking operation, a calling thread is suspended, and the check is suspended until the value maintained by the entity has reached or exceeded a given value.

2. A method as in claim 1 which said entity is allowed only to increment between allowable values, and not to decrement its value.

3. A method as in claim 1 wherein said entity is a counter that is only allowed to include integers.

4. A method as in claim 3 wherein an initial value of the counter is zero.

5. A method as in claim 1, wherein said entity is a/are flags.

1 6. An apparatus comprising a machine-readable storage
2 medium having executable instructions for managing threads in a
3 multithreaded system, the instructions enabling the machine to:
4 define an entity which maintains a count of values and
5 which is allowed to increment between allowable values;
6 determine a request for value of the element from a calling
7 thread; and
8 establish a check operation for said element in which said
9 calling thread is suspended until the entity reaches a
10 predetermined value.

1 7. An apparatus as in claim 6, wherein said entity is a
2 monotonically increasing counter.

1 8. An apparatus as in claim 6, wherein said entity is a
2 flag.

1 9. A apparatus as in claim 6 wherein said system has a
2 plurality of processors therein, wherein each of said processors
3 is running at least one different ones of said threads.

1 10. A method as in claim 1, further comprising defining an
2 error for an operation that decreases the value maintained by

3 the object to occur concurrently with any check operation on the
4 object.

1 11. A method as in claim 1, wherein the value maintained
2 by the object is a numeric value and the increment operation
3 increases the value by a numeric amount.

1 12. A method as in claim 1, wherein the value maintained
2 by the object is a Boolean value or a binary value and the
3 increment operation is a "set" operation that changes the value
4 from one state to the other state.

1 13. A method as in claim 2, wherein the value maintained
2 by the object is a Boolean value or a binary value and the
3 increment operation is a "set" operation that changes the value
4 from one state to the other state.

1 14. A method as in claim 12, further comprising
2 establishing an error for an increment operation on the object
3 to occur more than once.

1 15. A method of defining program code, comprising:
2 determining different parts of a program which can be
3 executed either sequentially, or in multithreaded parallel by

4 different threads, and which has equivalent results when
5 executed in said sequential or multithreaded parallel; and
6 defining said different parts as being multithreadable.

1 16. A method as in claim 15 wherein said determining is
2 based on a set of conditions that are sufficient to ensure the
3 equivalence of sequential and multithreaded execution of a
4 program construct.

1 17. A method as in claim 15 wherein said different parts
2 are defined as being multithreadable using an equivalence
3 annotation within the program code.

1 18. A method as in claim 17 wherein said annotation is a
2 pragma.

1 19. A method as in claim 17 wherein said annotation is a
2 code comment.

1 20. A method as in claim 15 further comprising, within
2 said code, multithreaded constructs, in addition to said
3 multithreadable parts.

1 21. A method as in claim 15 wherein said multithreadable
2 parts includes information which, if executed as threads, will
3 include the same result as if executed sequentially.

1 22. A method as in claim 15 wherein said part is a
2 multithreadable block of information.

1 23. A method as in claim 22 wherein said part is a
2 multithreadable FOR loop.

1 24. A method as in claim 15 further comprising
2 synchronizing threads using a monotonically-increasing counter.

1 25. A method as in claim 15 further comprising
2 synchronizing threads using a flag.

1 26. A method as in claim 16, wherein the equivalence
2 annotation includes a new or existing keyword or reserved word
3 in the program.

1 27. A method as in claim 16, wherein the equivalence
2 annotation takes the form of a character formatting in the
3 program, which can be such as boldface, italics, underlining, or
4 other formatting.

1 28. A method as in claim 16, wherein the equivalence
2 annotation takes the form of a special character sequence in the
3 program.

1 29. A method as in claim 16, wherein the equivalence
2 annotation is contained in a file or other entity separate from
3 the program.

1 30. A method as in claim 16, wherein the sequential
2 interpretation of the execution of the block construct is that
3 statements are executed one at a time in their textual order,
4 and the multithreaded interpretation of the execution of the
5 block construct is that statements of are partitioned among a
6 set of threads and executed concurrently by those threads.

1 31. A method as in claim 16 further comprising using
2 monotonic thread synchronization to synchronize actions among
3 threads.

1 32. A method as in claim 15 wherein:
2 explicitly multithreaded program constructs are always
3 executed according to a multithreaded interpretation

1 multithreadable program constructs are either executed
2 according a multithreaded interpretation or executed according
3 to a sequential interpretation; and
4 sequential or multithreaded execution of multithreadable
5 program constructs is at user selection.

1 33. A method as in claim 32, wherein the sequential or
2 multithreaded execution of multithreadable program constructs is
3 signalled by a pragma in the program.

1 34. A method as in claim 32, wherein the method for
2 selecting sequential or multithreaded execution of
3 multithreadable code constructs is a variable that is dependent
4 of the value of a variable defined in the program or in the
5 environment of the program.

1 35. A method of claim 32 wherein said multiple threaded
2 construct is a block or for loop.

1 36. A method of coding a program, comprising:
2 defining a first portion of code which must always be
3 executed according to multithreaded semantics, as a
4 multithreaded portion of code;

5 defining a second portion of code, within the same program
6 as said first portion of code, which may be selectively executed
7 according to either sequential or multithreaded techniques, as a
8 multithreadable code construct; and

9 allowing a program development system to develop said
10 multithreadable code construct as either a sequential or
11 multithreaded construct.

1 37. A method as in claim 36, wherein said program
2 development system includes a compiler.

1 38. A method as in claim 36 wherein said multithreaded
2 construct defines an operation which has no sequential
3 equivalent.

1 39. A method as in claim 38 wherein said multithreaded
2 construct is control of multiple windows in a graphical system.

1 40. A method as in claim 38 wherein said multithreaded
2 construct is control of different operations of a computer.

1 41. A method as in claim 37 wherein said operation is
2 executed on a multiple processor system, and different parts of
3 said operation are executed on different ones of the processors.

1 42. A method as in claim 37 wherein said multithreadable
2 constructs include a synchronization mechanism.

1 43. A method as in claim 42 wherein said synchronization
2 mechanism is a monotonically increasing counter.

1 44. A method as in claim 43 wherein said synchronization
2 mechanism is a special flag.

1 45. A method of integrating a structured multithreading
2 program development system with a standard program development
3 system, comprising:
4 detecting program elements which include a specified
5 annotation;
6 calling a special program development system element which
7 includes a processor that modifies based on the annotation to
8 form a preprocessed file; and
9 calling the standard program development system to compile
10 the preprocessed file.

1 46. A method of operating a program language, comprising:
2 defining equivalence annotations within the programming
3 language which indicate to a program development system of the

4 programming language information about sequential execution of
5 said statement; and

6 developing the programs as a sequential execution or as a
7 substantially simultaneous execution based on contents of the
8 equivalence annotations.

1 47. A method as in claim 46 wherein the equivalence
2 annotation indicates that the statements are multithreadable.

1 48. A method as in claim 46 wherein the equivalence
2 annotation indicates that the statements are either
3 multithreaded or multithreadable.

1 49. A method as in claim 48 wherein said multithreaded
2 statements must be executed in a multithreaded manner.

1 50. A method as in claim 48 wherein said multithreadable
2 annotations indicate that the statements can be executed in
3 either multithreaded or sequential manner.

1 51. A method as in claim 46 wherein said equivalence
2 annotation is a pragma.

1 52. A method as in claim 46 wherein said equivalence
2 annotation is a specially-defined comment line.

1 53. A method as in claim 47 further comprising
2 synchronizing access of threads to shared memory using a
3 specially defined synchronization element.

1 54. A method as in claim 53 wherein said synchronization
2 element is a synchronization counter.

1 55. A method as in claim 54 wherein said synchronization
2 counter is monotonically increasing, cannot be decreased, and
3 prevents thread operation during its check operation.

1 56. A method as in claim 53 wherein said synchronization
2 element is a synchronization flag.

1 57. A method as in claim 56 wherein said synchronization
2 counter is monotonically increasing, cannot be decreased, and
3 prevents thread operation during its check operation.

1 58. A method as in claim 54 wherein said s counter
2 includes a check operation, wherein said check operation
3 suspends a calling thread.

1 59. A method as in claim 58 further comprising maintaining
2 a list of suspended threads.

1 60. A method of modifying an existing program development
2 system and environment, comprising:

3 · detecting which components of a program contain
4 multithreadable program constructs or explicitly multithreaded
5 program constructs;

6 · transforming the components of the program that contain
7 multithreadable program constructs or explicitly multithreaded
8 program constructs into equivalent multithreaded components in a
9 form that can be directly translated or executed by the existing
10 program development system; and

11 invoking the existing program development system to
12 translate or execute the transformed components of the program.

1 61. A method as in claim 60, wherein said indicating
2 comprises giving distinctive names to said component.

1 62. A method as in claim 59, wherein the transforming of
2 the components of the program that contain multithreadable
3 program constructs or explicitly multithreaded program
4 constructs is by source-to-source program preprocessing.

1 63. A method as in claim 61, wherein the result of the
2 source-to-source program preprocessing is a program component
3 that incorporates thread library calls representing to the
4 transformed multithreadable program constructs or explicitly
5 multithreaded program constructs.

1 64. A method as in claim 63, wherein the thread library is
2 a thread library designed in part or whole for the purpose of
3 representing the transformed multithreadable program constructs
4 or explicitly multithreaded program constructs.

1 65. A method as in claim 63, wherein the thread library is
2 an existing thread library or a thread library designed for
3 another purpose.

1 66. A method as in claim 61, wherein the result of the
2 source-to-source program preprocessing is a program component
3 that incorporates standard multithreaded program constructs
4 supported by the existing programming system.

1 67. A method as in claim 59, further comprising renaming
2 the standard compiler-linker and the standard compiler-linker

3 name is used for a program component transformation tool that
4 subsequently invokes the renamed standard compiler-linker.

1 68. A method as in claim 59, wherein the operating system
2 is Linux or another variant of the Unix operating system and the
3 existing program development environment is the GNU C or C++
4 compiler or any other C or C++ compiler that operates under the
5 given variant of the Unix operating system.

1 69. A method as in claim 59, wherein the existing
2 programming language is a variant of the Java programming
3 language and the thread library is the standard Java thread
4 library.

1 70. A method of operating a program operation, comprising:
2 defining a block of code which can be executed either
3 sequentially or substantially simultaneously via separate loci
4 of execution;

5 running the program during a first mode in said sequential
6 mode, and running the program during a second mode in said
7 substantially simultaneous mode.

1 71. A method as in claim 70 wherein said definition is an
2 equivalence annotation.

1 72. A method as in claim 71 wherein said equivalence
2 annotation is a pragma.

1 73. A method as in claim 70 wherein, during said
2 sequential execution, variables are shared.

1 74. A method as in claim 73 wherein said shared variables
2 can be checked, and operation of check does not suspend
3 operations of the program.

1 75. A method as in claim 70 wherein during said
2 substantially simultaneous operations, variables are shared.

1 76. A method as in claim 70 further comprising debugging a
2 program in said sequential mode and running a debugged program
3 in said substantially simultaneous mode.

1 77. An object for synchronizing among multiple threads,
2 comprising:

3 a special object constrained to have (1) an integer
4 attribute value, (2) an increment function, but no decrement
5 function, and (3) check function that suspends a calling thread.

1 78. A method as in claim 77 wherein said check function
2 suspends a calling thread for a specified time.

1 79. An object as in claim 78 wherein said object includes
2 a list of thread suspension queues.

1 80. An object as in claim 77 further comprising a reset
2 function.

1 81. An object as in claim 77 wherein said object is a
2 counter.

1 82. An object as in claim 77 wherein said object is a flag
2 having only first and second values.

1 83. A method of integrating a thread management system
2 with an existing program development system, comprising:

3 first, running a pre-program development system that looks
4 for special annotations which indicate multithreaded and
5 multithreadable block of code;

6 using said special layer as an initial linker; and

7 then, passing the already linked program to the standard
8 program development system.

1 84. A method as in claim 83 wherein said program is a C
2 programming language.

11/11/2011 11:11:11 AM

Abstract

A structured multithreaded programming system is described for integrated use with existing and new programming languages and systems. The structured multithreaded programming system enables programs to be developed which include both multithreaded and multithreadable code constructs. The multithreaded code constructs require explicitly concurrent execution. The multithreadable code constructs can be executed either sequentially or concurrently, at the selection of the programmer or computer user. When executed concurrently, the different threads of execution in a multithreaded program developed with this system can be synchronized using innovative synchronization objects. One type of synchronization object is a special type of counter, which can be constrained to be monotonically increasing in value. Another related type of synchronization object is a special type of flag, which can be constrained to have its value set monotonically.

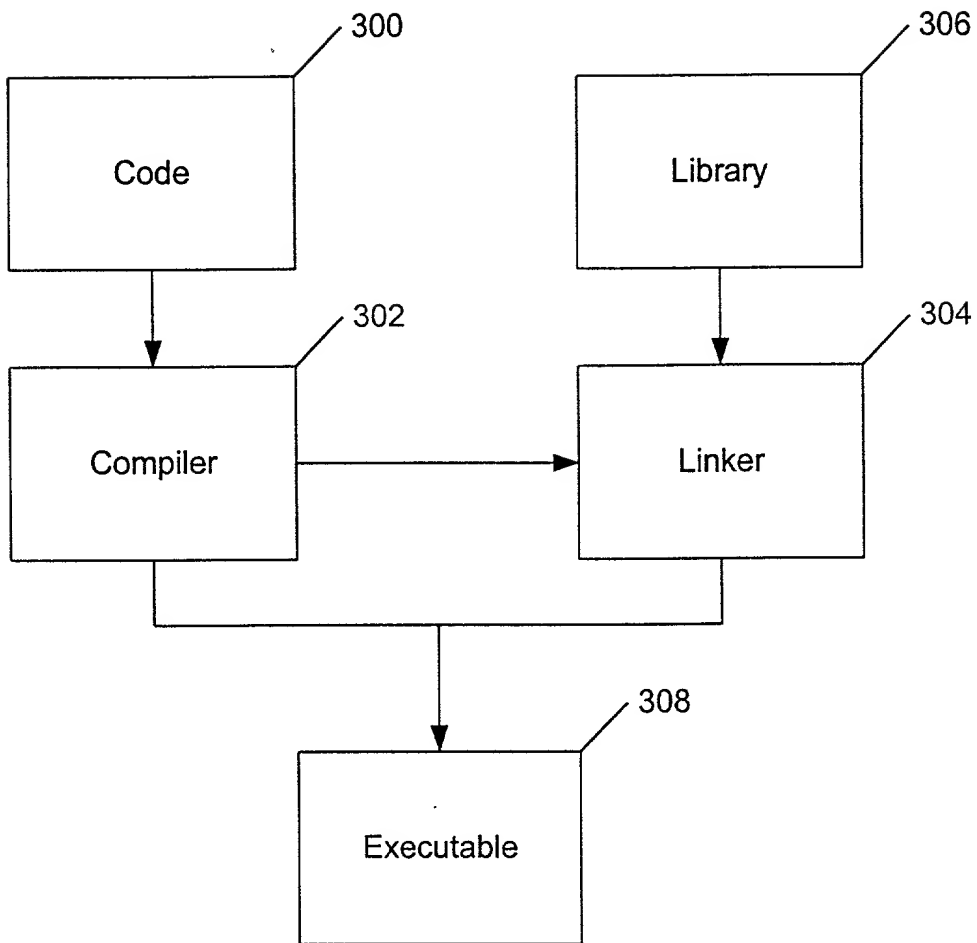
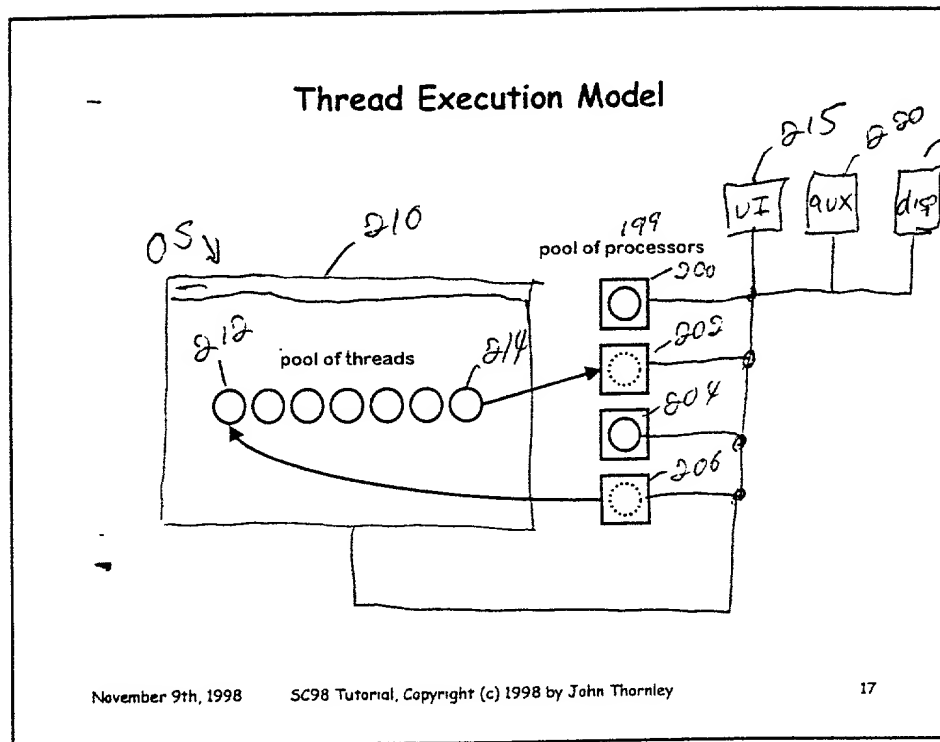


FIG. 1
(Prior Art)

FIG 2



Thread Execution Model: Key Points

- Pool of processors, pool of threads.
- Threads are peers.
- Dynamic thread creation.
- Can support many more threads than processors.
- Threads dynamically switch between processors.
- Threads share access to memory.
- Synchronization needed between threads.

November 9th, 1998 SC98 Tutorial, Copyright (c) 1998 by John Thornley 18

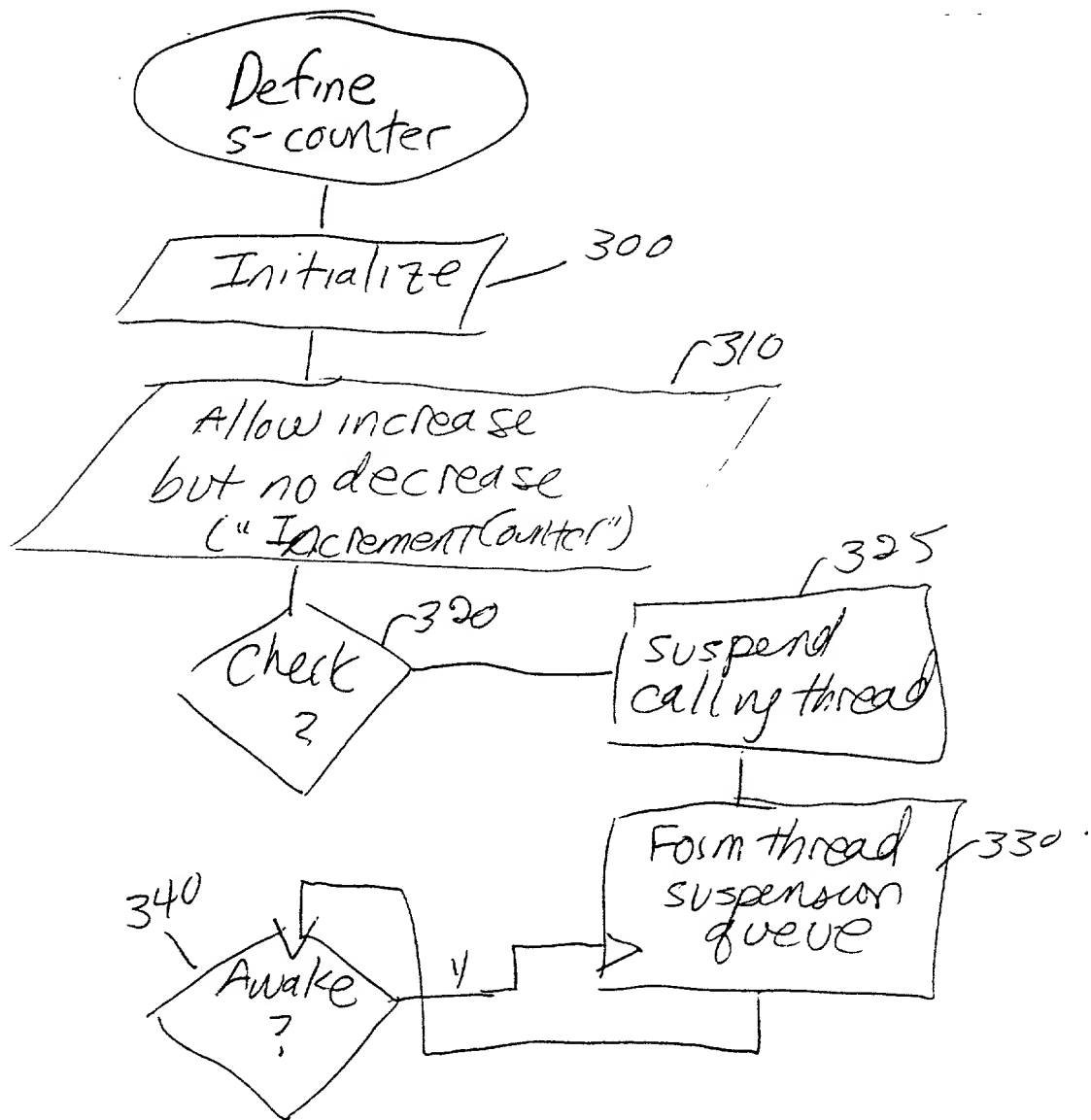


FIG 3

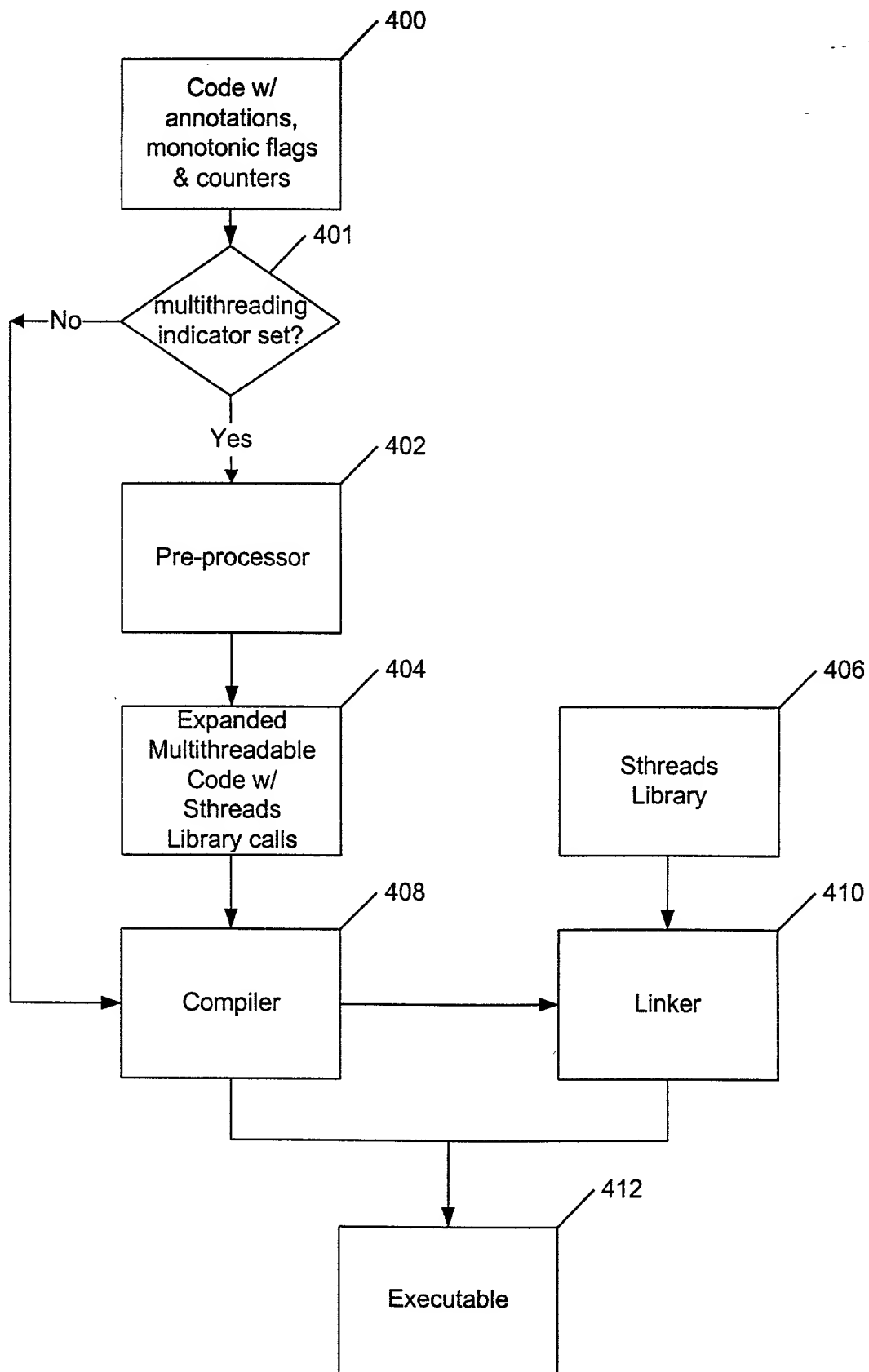


FIG. 24

Implementation of the Sthreads library.

```

#ifndef STHREADS_H
#define STHREADS_H

#ifdef _WIN32
#error ERROR: Win32 pthreads.h included in non-Win32 program.
#endif

#ifdef _MT
#error ERROR: Sthreads program must be linked with multithreaded libraries.
#endif

#ifdef __cplusplus
extern "C" {
#endif

/*-----*/
/* Sthreads: A Structured Thread Library for Shared-Memory Multiprocessing */
/* Version 1.0 for Windows NT */
/*
/* Author: John Thornley, Computer Science Dept., Caltech.
/* Date: September 1998.
/*
/* Copyright (c) 1998 by John Thornley.
/*-----*/

/*-----*/
/* Error codes
/*-----*/

#define STHREADS_ERROR_NONE 0
#define STHREADS_ERROR_INPUTVALUE 1
#define STHREADS_ERROR_MEMORYALLOC 2
#define STHREADS_ERROR_THREADCREATE 3
#define STHREADS_ERROR_SYNC_CREATE 4
#define STHREADS_ERROR_UNINITIALIZED 5
#define STHREADS_ERROR_FINALIZED 6
#define STHREADS_ERROR_INUSE 7
#define STHREADS_ERROR_LOCKHELD 8
#define STHREADS_ERROR_LOCKNOTHELD 9
#define STHREADS_ERROR_COUNTEROVERFLOW 10
#define STHREADS_ERROR_UNSPECIFIED 11
#define STHREADS_ERROR_UNSPECIFIED 12
#define STHREADS_ERROR_UNSPECIFIED 12

/* Requirements:
/* - STHREADS_ERROR_NONE == 0.
/* - STHREADS_ERROR_INPUTVALUE > STHREADS_ERROR_NONE.
/* - STHREADS_ERROR_MEMORYALLOC > STHREADS_ERROR_INPUTVALUE.
/* - STHREADS_ERROR_THREADCREATE > STHREADS_ERROR_MEMORYALLOC.
/* - STHREADS_ERROR_SYNC_CREATE > STHREADS_ERROR_THREADCREATE.
/* - STHREADS_ERROR_UNINITIALIZED > STHREADS_ERROR_SYNC_CREATE.
/* - STHREADS_ERROR_FINALIZED > STHREADS_ERROR_UNINITIALIZED.
/* - STHREADS_ERROR_INUSE > STHREADS_ERROR_FINALIZED.
/* - STHREADS_ERROR_LOCKHELD > STHREADS_ERROR_INUSE.
/* - STHREADS_ERROR_LOCKNOTHELD > STHREADS_ERROR_LOCKHELD.
/* - STHREADS_ERROR_COUNTEROVERFLOW > STHREADS_ERROR_LOCKNOTHELD.
/* - STHREADS_ERROR_UNSPECIFIED > STHREADS_ERROR_COUNTEROVERFLOW.
/* - STHREADS_ERROR_UNSPECIFIED < INT_MAX.

/*-----*/
/* Error string maximum length
/*-----*/

#define STHREADS_ERROR_STRING_MAX 100

/* Requirements:
/* - STHREADS_ERROR_STRING_MAX >= 1.
/* - STHREADS_ERROR_STRING_MAX <= INT_MAX.

/*-----*/
/* Processors
/*-----*/

#define STHREADS_PROCESSORS_MAX 32

```

```

#define STHREADS_PROCESSOR_YES 1000
#define STHREADS_PROCESSOR_NO 1001

/* Requirements: */
/* - STHREADS_PROCESSORS_MAX >= 1. */
/* - STHREADS_PROCESSORS_MAX <= INT_MAX. */
/* - STHREADS_PROCESSOR_YES >= INT_MIN. */
/* - STHREADS_PROCESSOR_YES <= INT_MAX. */
/* - STHREADS_PROCESSOR_NO >= INT_MIN. */
/* - STHREADS_PROCESSOR_NO <= INT_MAX. */
/* - STHREADS_PROCESSOR_YES != STHREADS_PROCESSOR_NO. */

/* Definitions: */
/* - ValidProcessorStatus(p) = */
/*     p == STHREADS_PROCESSOR_PRESENT || */
/*     p == STHREADS_PROCESSOR_NOT_PRESENT. */

/*-----*/
/* Mappings of statements/iterations to threads */
/*-----*/

#define STHREADS_MAPPING_SIMPLE 3000
#define STHREADS_MAPPING_DYNAMIC 3001
#define STHREADS_MAPPING_BLOCKED 3002
#define STHREADS_MAPPING_INTERLEAVED 3003

/* Requirements: */
/* - STHREADS_MAPPING_SIMPLE > 0. */
/* - STHREADS_MAPPING_DYNAMIC == STHREADS_MAPPING_SIMPLE + 1. */
/* - STHREADS_MAPPING_BLOCKED == STHREADS_MAPPING_DYNAMIC + 1. */
/* - STHREADS_MAPPING_INTERLEAVED == STHREADS_MAPPING_BLOCKED + 1. */
/* - STHREADS_MAPPING_INTERLEAVED < INT_MAX. */

/* Definitions: */
/* - ValidMapping(m) = */
/*     m == STHREADS_MAPPING_SIMPLE || */
/*     m == STHREADS_MAPPING_DYNAMIC || */
/*     m == STHREADS_MAPPING_BLOCKED || */
/*     m == STHREADS_MAPPING_INTERLEAVED. */

/*-----*/
/* Conditions testable in regular for loop control */
/*-----*/

#define STHREADS_CONDITION_LT 4000
#define STHREADS_CONDITION_LE 4001
#define STHREADS_CONDITION_GT 4002
#define STHREADS_CONDITION_GE 4003

/* Requirements: */
/* - STHREADS_CONDITION_LT > 0. */
/* - STHREADS_CONDITION_LE == STHREADS_CONDITION_LT + 1. */
/* - STHREADS_CONDITION_GT == STHREADS_CONDITION_LE + 1. */
/* - STHREADS_CONDITION_GE == STHREADS_CONDITION_GT + 1. */
/* - STHREADS_CONDITION_GE < INT_MAX. */

/* Definitions: */
/* - ValidCondition(c) = */
/*     c == STHREADS_CONDITION_LT || */
/*     c == STHREADS_CONDITION_LE || */
/*     c == STHREADS_CONDITION_GT || */
/*     c == STHREADS_CONDITION_GE. */

/*-----*/
/* Stack sizes (in bytes) */
/*-----*/

#define STHREADS_STACK_SIZE_MINIMUM 16384
#define STHREADS_STACK_SIZE_DEFAULT 262144

/* Requirements: */
/* - STHREADS_STACK_SIZE_MINIMUM >= 0. */
/* - STHREADS_STACK_SIZE_DEFAULT >= STHREADS_STACK_SIZE_MINIMUM. */
/* - STHREADS_STACK_SIZE_DEFAULT <= UINT_MAX. */

```



```

/* Definitions: */
/* - ValidStackSize(s) = */
/*     s >= STHREADS_STACK_SIZE_MINIMUM. */

/*-----*/
/* Priorities */
/*-----*/

#define STHREADS_PRIORITY_LOWEST -2
#define STHREADS_PRIORITY_HIGHEST +2
#define STHREADS_PRIORITY_PARENT 10000 /* Inherit priority of parent thread. */

/* Requirements: */
/* - STHREADS_PRIORITY_LOWEST > INT_MIN. */
/* - STHREADS_PRIORITY_HIGHEST >= STHREADS_PRIORITY_LOWEST. */
/* - STHREADS_PRIORITY_HIGHEST < INT_MAX. */
/* - STHREADS_PRIORITY_PARENT < STHREADS_PRIORITY_LOWEST || */
/*   STHREADS_PRIORITY_PARENT > STHREADS_PRIORITY_HIGHEST. */

/* Definitions: */
/* - ValidPriority(p) = */
/*     STHREADS_PRIORITY_LOWEST <= p && p <= STHREADS_PRIORITY_HIGHEST. */

/*-----*/
/* Print error message to string */
/*-----*/

void SthreadsWriteErrorMessage(int errorCode, char errorString[]);

/* Input Arguments: */
/* - errorCode : error code returned by an Sthreads function call. */
/* Output Arguments: */
/* - errorString : error message as a char string. */
/* Preconditions: */
/* - errorString != NULL && */
/*   errorString is a string of at least STHREADS_ERROR_STRING_MAX chars. */
/* Postconditions: */
/* - errorString is '\0' terminated string of chars in the range ' ' .. '~'. */
/* - 1 <= strlen(errorString) < STHREADS_ERROR_STRING_MAX. */
/* Atomicity: */
/* - Atomic with respect to all operations. */

/*-----*/
/* Handle errors. */
/*-----*/

void SthreadsErrorHandler(int errorCode);

/* Input Arguments: */
/* - errorCode : error code returned by an Sthreads function call. */
/* Operation: */
/* - error handler function is called with errorCode as argument. */
/* Default Error Handler Function: */
/* - Displays error message and terminates normal program execution. */
/* Atomicity: */
/* - Not atomic with respect to SthreadsSetErrorHandler operations. */
/* - Atomic with respect to all other operations. */

/*-----*/
/* Set error handler function. */
/*-----*/

int SthreadsSetErrorHandler(void (*errorHandler)(int errorCode));

/* Input Arguments: */
/* - errorHandler : function to handle errors. */
/* Preconditions: */
/* - errorHandler == NULL || */
/*   errorHandler is valid void (*)(int) function. */
/* Postconditions: */
/* - if (errorHandler == NULL) */
/*   error handler function is set to default error handler function. */
/* - if (errorHandler != NULL)

```

```

/*      error handler function is set to ErrorHandler.          */
/* Atomicity:                                                    */
/* - Not atomic with respect to                                  */
/*   SthreadsHandleError and SthreadsSetErrorHandler operations. */
/* - Atomic with respect to all other operations.                */

/*-----*/
/* Control the processors used by program execution.             */
/*-----*/

int SthreadsGetSystemProcessors(int processor[]);

/* Output Arguments:                                             */
/* - processors : processors that exist on the system.           */
/* Function Return:                                              */
/* - error code.                                                 */
/* Preconditions:                                                */
/* - processor != NULL &&                                         */
/*   processor is an array of at least STHREADS_PROCESSORS_MAX ints. */
/* Postconditions:                                               */
/* - forall (p = 0; p < STHREADS_PROCESSORS_MAX; p++)            */
/*   ValidProcessorStatus(processor[p]) &&                       */
/*   (if (processor[p] == STHREADS_PROCESSOR_YES)                */
/*     a processor numbered p exists on the system) &&          */
/*   (if (processor[p] == STHREADS_PROCESSOR_NO)                */
/*     a processor numbered p does not exist on the system).     */
/* Atomicity:                                                    */
/* - Atomic with respect to all operations.                      */

int SthreadsSetProgramProcessors(int processor[]);

/* Input Arguments:                                             */
/* - processor : processors on which the threads of the program may execute. */
/* Function Return:                                              */
/* - error code.                                                 */
/* Preconditions:                                                */
/* - processor != NULL &&                                         */
/*   processor is an array of at least STHREADS_PROCESSORS_MAX ints. */
/* - forall (p = 0; p < STHREADS_PROCESSORS_MAX; p++)            */
/*   ValidProcessorStatus(processor[p]) &&                       */
/*   if (processor[p] == STHREADS_PROCESSOR_YES)                */
/*     a processor numbered p exists on the system.             */
/* - exists (p = 0; p < STHREADS_PROCESSORS_MAX; p++)            */
/*   processor[p] == STHREADS_PROCESSOR_YES.                     */
/* Atomicity:                                                    */
/* - Must be called when program execution consists of a single thread.

int SthreadsGetProgramProcessors(int processor[]);

/* Output Arguments:                                             */
/* - processors : processors on which the program may execute.   */
/* Function Return:                                              */
/* - error code.                                                 */
/* Preconditions:                                                */
/* - processor != NULL &&                                         */
/*   processor is an array of at least STHREADS_PROCESSORS_MAX ints. */
/* Postconditions:                                               */
/* - forall (p = 0; p < STHREADS_PROCESSORS_MAX; p++)            */
/*   ValidProcessorStatus(processor[p]) &&                       */
/*   (if (processor[p] == STHREADS_PROCESSOR_YES)                */
/*     the program may execute on processor number p) &&          */
/*   (if (processor[p] == STHREADS_PROCESSOR_NO)                */
/*     the program may not execute on processor number p).       */
/* Atomicity:                                                    */
/* - Not atomic with respect to                                  */
/*   SetProgramProcessors and SetNumProgramProcessors operations. */
/* - Atomic with respect to all other operations.

int SthreadsSetThreadProcessors(int processor[]);

/* Input Arguments:                                             */
/* - processor : processors on which the thread may execute.     */
/* Function Return:                                              */
/* - error code.

```

```

/* Preconditions: */
/* - processor != NULL && */
/* processor is an array of at least STHREADS_PROCESSORS_MAX ints. */
/* - forall (p = 0; p < STHREADS_PROCESSORS_MAX; p++) */
/*     ValidProcessorStatus(processor[p]) && */
/*     if (processor[p] == STHREADS_PROCESSOR_YES) */
/*         the program may execute on processor number p. */
/* - exists (p = 0; p < STHREADS_PROCESSORS_MAX; p++) */
/*     processor[p] == STHREADS_PROCESSOR_YES. */
/* Atomicity: */
/* - Not atomic with respect to */
/* SetProgramProcessors and SetNumProgramProcessors operations. */
/* - Atomic with respect to all other operations. */

int SthreadsGetNumSystemProcessors(int *numProcessors);

/* Output Arguments: */
/* - numProcessors : number of processors that exist on the system. */
/* Function Return: */
/* - error code. */
/* Preconditions: */
/* - numProcessors != NULL && numProcessors points to a valid int variable. */
/* Postconditions: */
/* - *numProcessors == number of processors that exist on the system. */
/* Atomicity: */
/* - Atomic with respect to all operations. */

int SthreadsSetNumProgramProcessors(int numProcessors);

/* Input Arguments: */
/* - numProcessors : number of processors on which the threads of the program */
/* may execute. */
/* Function Return: */
/* - error code. */
/* Preconditions: */
/* - numProcessors >= 1. */
/* - numProcessors <= number of processors that exist on the system. */
/* Atomicity: */
/* - Must be called when program execution consists of a single thread. */
/* ----- */
/* Multithreaded block */
/* ----- */

int SthreadsBlock(
    int numStatements, void (*statement[])(void *args), void *args,
    int mapping, int numThreads,
    int priority, unsigned int stackSize);

/* Input Arguments: */
/* - numStatements : number of statements in block. */
/* - statement : functions representing statements. */
/* - args : pointer to arguments of the statements. */
/* - mapping : mapping of statements onto threads. */
/* - numThreads : number of threads. */
/* - priority : priority of threads. */
/* - stackSize : stack size of threads. */
/* Function Return: */
/* - error code. */
/* Preconditions: */
/* - numStatements >= 0. */
/* - statement != NULL && */
/* statement is an array of at least numStatements functions. */
/* - forall (s = 0; s < numStatements; s++) */
/*     statement[s] != NULL && */
/*     statement[s] is a valid void (*) (void *) function. */
/* - ValidMapping(mapping). */
/* - if (mapping != STHREADS_MAPPING_SIMPLE) */
/*     (numThreads > 0) || (numThreads == 0 && numStatements == 0). */
/* - ValidPriority(priority) || priority == STHREADS_PRIORITY_PARENT. */
/* - ValidStackSize(stackSize). */
/* Atomicity: */
/* - Atomic with respect to all operations. */

```

```

/*-----*/
/* Multithreaded regular for loop */
/*-----*/

int SthreadsRegularForLoop(
    void (*chunk)(int initial, int bound, int step, void *args), void *args,
    int initial, int condition, int bound, int step,
    int chunkSize, int mapping, int numThreads,
    int priority, unsigned int stackSize);

/* Input Arguments: */
/* - chunk      : function to execute iterations of loop body. */
/* - args       : pointer to arguments of loop body. */
/* - initial     : initial value of control variable. */
/* - condition   : condition between control variable and bound value. */
/* - bound      : bound value of control variable. */
/* - step       : step value of control variable. */
/* - chunkSize  : number of iterations per chunk. */
/* - mapping     : mapping of chunks onto threads. */
/* - numThreads : number of threads. */
/* - priority   : priority of threads. */
/* - stackSize  : stack size of threads. */
/* Function Return: */
/* - error code. */
/* Preconditions: */
/* - chunk != NULL && */
/*   chunk is a valid void (*)(int, int, int, void *) function. */
/* - ValidCondition(condition). */
/* - !InfiniteRange(initial, condition, bound, step). */
/* - (chunkSize > 0) || */
/*   (chunkSize == 0 && NullRange(initial, condition, bound, step)). */
/* - ValidMapping(mapping). */
/* - if (mapping != STHREADS_MAPPING_SIMPLE) */
/*   (numThreads > 0) || */
/*   (numThreads == 0 && NullRange(initial, condition, bound, step)). */
/* - ValidPriority(priority) || priority == STHREADS_PRIORITY_PARENT. */
/* - ValidStackSize(stackSize). */

/* Definitions: */
/* - InfiniteRange(initial, condition, bound, step) = */
/*   (condition == STHREADS_CONDITION_LT && */
/*     initial < bound && step <= 0) || */
/*   (condition == STHREADS_CONDITION_LE && */
/*     initial <= bound && step <= 0) || */
/*   (condition == STHREADS_CONDITION_GT && */
/*     initial > bound && step >= 0) || */
/*   (condition == STHREADS_CONDITION_GE && */
/*     initial >= bound && step >= 0). */
/* - NullRange(initial, condition, bound, step) = */
/*   (condition == STHREADS_CONDITION_LT && initial >= bound) || */
/*   (condition == STHREADS_CONDITION_LE && initial > bound) || */
/*   (condition == STHREADS_CONDITION_GT && initial <= bound) || */
/*   (condition == STHREADS_CONDITION_GE && initial < bound). */
/* Atomicity: */
/* - Atomic with respect to all operations.

/*-----*/
/* Flags */
/*-----*/

typedef struct {
    unsigned char value[16];
} SthreadsFlag;

int SthreadsFlagInitialize(SthreadsFlag *flag);

/* Input-Output Arguments: */
/* - flag : flag variable. */
/* Function Return: */
/* - error code. */
/* Preconditions: */
/* - flag != NULL && flag points to a valid flag variable. */
/* - !Initialized(flag). */
/* Atomicity:

```

```

/* - Not atomic with respect to all other operations on flag.          */
/* - Atomic with respect to all other operations.                        */

int SthreadsFlagFinalize(SthreadsFlag *flag);

/* Input-Output Arguments:                                             */
/* - flag : flag variable.                                             */
/* Function Return:                                                    */
/* - error code.                                                        */
/* Preconditions:                                                       */
/* - flag != NULL && flag points to a valid flag variable.            */
/* - Initialized(flag) && !Finalized(flag).                             */
/* - NumWaiting(flag) == 0.                                             */
/* Atomicity:                                                            */
/* - Not atomic with respect to all other operations on flag.          */
/* - Atomic with respect to all other operations.                       */

int SthreadsFlagSet(SthreadsFlag *flag);

/* Input-Output Arguments:                                             */
/* - flag : flag variable.                                             */
/* Function Return:                                                    */
/* - error code.                                                        */
/* Preconditions:                                                       */
/* - flag != NULL && flag points to a valid flag variable.            */
/* - Initialized(flag) && !Finalized(flag).                             */
/* Atomicity:                                                            */
/* - Atomic with respect to Set and Check operations on flag.          */
/* - Not atomic with respect to other operations on flag.              */
/* - Atomic with respect to all other operations.                       */

int SthreadsFlagCheck(SthreadsFlag *flag);

/* Input-Output Arguments:                                             */
/* - flag : flag variable.                                             */
/* Function Return:                                                    */
/* - error code.                                                        */
/* Preconditions:                                                       */
/* - flag != NULL && flag points to a valid flag variable.            */
/* - Initialized(flag) && !Finalized(flag).                             */
/* Atomicity:                                                            */
/* - Atomic with respect to Set and Check operations on flag.          */
/* - Not atomic with respect to other operations on flag.              */
/* - Atomic with respect to all other operations.                       */

int SthreadsFlagReset(SthreadsFlag *flag);

/* Input-Output Arguments:                                             */
/* - flag : flag variable.                                             */
/* Function Return:                                                    */
/* - error code.                                                        */
/* Preconditions:                                                       */
/* - flag != NULL && flag points to a valid flag variable.            */
/* - Initialized(flag) && !Finalized(flag).                             */
/* - NumWaiting(flag) == 0.                                             */
/* Atomicity:                                                            */
/* - Not atomic with respect to other operations on flag.              */
/* - Atomic with respect to all other operations.                       */

/*-----*/
/* Counters                                                             */
/*-----*/

typedef struct {
    unsigned char value[40];
} SthreadsCounter;

int SthreadsCounterInitialize(SthreadsCounter *counter);

/* Input-Output Arguments:                                             */
/* - counter : pointer to counter variable.                             */
/* Function Return:                                                    */
/* - error code.                                                        */
/* Preconditions:                                                       */

```

```

/* - counter != NULL && counter points to a valid counter variable. */
/* - !Initialized(counter). */
/* Atomicity: */
/* - Not atomic with respect to all other operations on counter. */
/* - Atomic with respect to all other operations. */

int SthreadsCounterFinalize(SthreadsCounter *counter);

/* Input-Output Arguments: */
/* - counter : pointer to counter variable. */
/* Function Return: */
/* - error code. */
/* Preconditions: */
/* - counter != NULL && counter points to a valid counter variable. */
/* - Initialized(counter) && !Finalized(counter). */
/* - NumWaiting(counter) == 0. */
/* Atomicity: */
/* - Not atomic with respect to all other operations on counter. */
/* - Atomic with respect to all other operations. */

int SthreadsCounterIncrement(SthreadsCounter *counter, unsigned int amount);

/* Input-Output Arguments: */
/* - counter : pointer to counter variable. */
/* Function Return: */
/* - error code. */
/* Preconditions: */
/* - counter != NULL && counter points to a valid counter variable. */
/* - Initialized(counter) && !Finalized(counter). */
/* - Count(counter) <= UINT_MAX - amount. */
/* Atomicity: */
/* - Atomic with respect to Increment and Check operations on counter. */
/* - Not atomic with respect to other operations on counter. */
/* - Atomic with respect to all other operations. */

int SthreadsCounterCheck(SthreadsCounter *counter, unsigned int value);

/* Input-Output Arguments: */
/* - counter : pointer to counter variable. */
/* Function Return: */
/* - error code. */
/* Preconditions: */
/* - counter != NULL && counter points to a valid counter variable. */
/* - Initialized(counter) && !Finalized(counter). */
/* Atomicity: */
/* - Atomic with respect to Increment and Check operations on counter. */
/* - Not atomic with respect to other operations on counter. */
/* - Atomic with respect to all other operations. */

int SthreadsCounterReset(SthreadsCounter *counter);

/* Input-Output Arguments: */
/* - counter : pointer to counter variable. */
/* Function Return: */
/* - error code. */
/* Preconditions: */
/* - counter != NULL && counter points to a valid counter variable. */
/* - Initialized(counter) && !Finalized(counter). */
/* - NumWaiting(counter) == 0. */
/* Atomicity: */
/* - Not atomic with respect to all other operations on counter. */
/* - Atomic with respect to all other operations. */

/*-----*/
/* Locks */
/*-----*/

typedef struct {
    unsigned char value[36];
} SthreadsLock;

int SthreadsLockInitialize(SthreadsLock *lock);

/* Input-Output Arguments: */

```

```

/* - lock : pointer to lock variable. */
/* Function Return: */
/* - error code. */
/* Preconditions: */
/* - lock != NULL && lock points to a valid lock variable. */
/* - !Initialized(lock). */
/* Atomicity: */
/* - Not atomic with respect to all other operations on lock. */
/* - Atomic with respect to all other operations. */

int SthreadsLockFinalize(SthreadsLock *lock);

/* Input-Output Arguments: */
/* - lock : pointer to lock variable. */
/* Function Return: */
/* - error code. */
/* Preconditions: */
/* - lock != NULL && lock points to a valid lock variable. */
/* - Initialized(lock) && !Finalized(lock). */
/* - !AnyThreadHolds(lock). */
/* Atomicity: */
/* - Not atomic with respect to all other operations on lock. */
/* - Atomic with respect to all other operations. */

int SthreadsLockAcquire(SthreadsLock *lock);

/* Input-Output Arguments: */
/* - lock : pointer to lock variable. */
/* Function Return: */
/* - error code. */
/* Preconditions: */
/* - lock != NULL && lock points to a valid lock variable. */
/* - Initialized(lock) && !Finalized(lock). */
/* - !ThisThreadHolds(lock). */
/* Atomicity: */
/* - Atomic with respect to Acquire and Release operations on lock. */
/* - Not atomic with respect to other operations on lock. */
/* - Atomic with respect to all other operations. */

int SthreadsLockRelease(SthreadsLock *lock);

/* Input-Output Arguments: */
/* - lock : pointer to lock variable. */
/* Function Return: */
/* - error code. */
/* Preconditions: */
/* - lock != NULL && lock points to a valid lock variable. */
/* - Initialized(lock) && !Finalized(lock). */
/* - ThisThreadHolds(lock). */
/* Atomicity: */
/* - Atomic with respect to Acquire and Release operations on lock. */
/* - Not atomic with respect to other operations on lock. */
/* - Atomic with respect to all other operations. */

/*-----*/
/* Barriers */
/*-----*/

typedef struct {
    unsigned char value[52];
} SthreadsBarrier;

int SthreadsBarrierInitialize(SthreadsBarrier *barrier, int numThreads);

/* Input-Output Arguments: */
/* - barrier : pointer to barrier variable. */
/* - numThreads : number of threads that cross barrier in each pass. */
/* Function Return: */
/* - error code. */
/* Preconditions: */
/* - barrier != NULL && barrier points to a valid barrier variable. */
/* - !Initialized(barrier). */
/* - numThreads >= 1. */
/* Atomicity: */

```

```

/* - Not atomic with respect to all other operations on barrier. */
/* - Atomic with respect to all other operations. */

int SthreadsBarrierFinalize(SthreadsBarrier *barrier);

/* Input-Output Arguments: */
/* - barrier : pointer to barrier variable. */
/* Function Return: */
/* - error code. */
/* Preconditions: */
/* - barrier != NULL && barrier points to a valid barrier variable. */
/* - Initialized(barrier) && !Finalized(barrier). */
/* - NumWaiting(barrier) == 0. */
/* Atomicity: */
/* - Not atomic with respect to all other operations on barrier. */
/* - Atomic with respect to all other operations. */

int SthreadsBarrierPass(SthreadsBarrier *barrier);

/* Input-Output Arguments: */
/* - barrier : pointer to barrier variable. */
/* Function Return: */
/* - error code. */
/* Preconditions: */
/* - barrier != NULL && barrier points to a valid barrier variable. */
/* - Initialized(barrier) && !Finalized(barrier). */
/* Atomicity: */
/* - Atomic with respect to Pass operations on barrier. */
/* - Not atomic with respect to other operations on barrier. */
/* - Atomic with respect to all other operations. */

int SthreadsBarrierReset(SthreadsBarrier *barrier, int numThreads);

/* Input-Output Arguments: */
/* - barrier : pointer to barrier variable. */
/* Function Return: */
/* - error code. */
/* Preconditions: */
/* - barrier != NULL && barrier points to a valid barrier variable. */
/* - Initialized(barrier) && !Finalized(barrier). */
/* - NumWaiting(barrier) == 0. */
/* - numThreads >= 1. */
/* Atomicity: */
/* - Not atomic with respect to all other operations on barrier. */
/* - Atomic with respect to all other operations. */

/*-----*/
/* Priorities */
/*-----*/

int SthreadsGetCurrentPriority(int *priority);

/* Output Arguments: */
/* - priority : scheduling priority of calling thread. */
/* Function Return: */
/* - error code. */
/* Preconditions: */
/* - priority != NULL && priority points to a valid int variable. */
/* Postconditions: */
/* - *priority == scheduling priority of calling thread. */
/* Atomicity: */
/* - Atomic with respect to all operations. */

int SthreadsSetCurrentPriority(int priority);

/* Input Arguments: */
/* - priority : scheduling priority for calling thread. */
/* Function Return: */
/* - error code. */
/* Preconditions: */
/* - ValidPriority(priority). */
/* Atomicity: */
/* - Atomic with respect to all operations. */

```



```

/*-----*/
#ifdef __cplusplus
}
#endif

#endif /* !STHEADS_H */

```

```

/*-----*/
/* Sthreads: A Structured Thread Library for Shared-Memory Multiprocessing */
/* Version 1.0 for Windows NT */
/*
/* Author: John Thornley, Computer Science Dept., Caltech.
/* Date: September 1998.
/*
/* Copyright (c) 1998 by John Thornley.
/*
/* THINGS TO DO:
/*
/* - Change names of CHECK tests, e.g., to CHECKNOTINITIALIZED.
/* - Make Finalize operations set Initialized and Finalized flags to false.
/* - Counter for dynamic for loop should be unsigned int.
/* - Declarations of thread functions should be compatible with
/* Win32 prototype ... see page 25.
/* - Implement special case of BarrierPass when numThreads == 1.
/* - Implement flags like counters for efficiency when flag is set?
/* - Change priority low and high to THREAD_PRIORITY_IDLE and _TIME_CRITICAL.
/*-----*/

#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <limits.h>
#include <windows.h>
#include "sthreads.h"

/*-----*/
/* Bool type definition */
/*-----*/
typedef int bool;
#define false 0
#define true 1

/*-----*/
/* Miscellaneous utility definitions */
/*-----*/
#define MIN(x, y) ((x) < (y) ? (x) : (y))
#define MAX(x, y) ((x) > (y) ? (x) : (y))

/*-----*/
/* Verify requirements, beliefs, and checks */
/*-----*/
#define require(condition) assert(condition) /* require this input condition */
#define believe(condition) assert(condition) /* believe this must be true */
#define check(condition) assert(condition) /* check this is true

/*-----*/
/* Check for error conditions */
/*-----*/

#define CHECKINPUTVALUE(condition) \
    if (!(condition)) { return STHREADS_ERROR_INPUTVALUE; }

#define CHECKMEMORYALLOC(condition) \
    if (!(condition)) { return STHREADS_ERROR_MEMORYALLOC; }

#define CHECKTHREADCREATE(condition) \
    if (!(condition)) { return STHREADS_ERROR_THREADCREATE; }

#define CHECKSYNCCREATE(condition) \
    if (!(condition)) { return STHREADS_ERROR_SYNCCREATE; }

#define CHECKINITIALIZED(condition) \
    if (!(condition)) { return STHREADS_ERROR_INITIALIZED; }

#define CHECKUNINITIALIZED(condition) \
    if (!(condition)) { return STHREADS_ERROR_UNINITIALIZED; }

```

```

#define CHECKFINALIZED(condition) \
    if (!(condition)) { return STHREADS_ERROR_FINALIZED; }

#define CHECKINUSE(condition) \
    if (!(condition)) { return STHREADS_ERROR_INUSE; }

#define CHECKLOCKHELD(condition) \
    if (!(condition)) { return STHREADS_ERROR_LOCKHELD; }

#define CHECKLOCKNOTHELD(condition) \
    if (!(condition)) { return STHREADS_ERROR_LOCKNOTHELD; }

#define CHECKCOUNTEROVERFLOW(condition) \
    if (!(condition)) { return STHREADS_ERROR_COUNTEROVERFLOW; }

#define CHECKOTHER(condition) \
    if (!(condition)) { return STHREADS_ERROR_UNSPECIFIED; }

/*-----*/
/* Is processor status value valid? */
/*-----*/

static bool ValidProcessorStatus(int p)
{
    return
        p == STHREADS_PROCESSOR_YES ||
        p == STHREADS_PROCESSOR_NO;
}

/*-----*/
/* Is mapping value valid? */
/*-----*/

static bool ValidMapping(int m)
{
    return
        m == STHREADS_MAPPING_SIMPLE ||
        m == STHREADS_MAPPING_DYNAMIC ||
        m == STHREADS_MAPPING_BLOCKED ||
        m == STHREADS_MAPPING_INTERLEAVED;
}

/*-----*/
/* Is condition value valid? */
/*-----*/

static bool ValidCondition(int c)
{
    return
        c == STHREADS_CONDITION_LT ||
        c == STHREADS_CONDITION_LE ||
        c == STHREADS_CONDITION_GT ||
        c == STHREADS_CONDITION_GE;
}

/*-----*/
/* Is stack-size value valid? */
/*-----*/

static bool ValidStackSize(unsigned int s)
{
    return
        s >= STHREADS_STACK_SIZE_MINIMUM;
}

/*-----*/
/* Is priority value valid? */
/*-----*/

static bool ValidPriority(int p)
{
    return
        STHREADS_PRIORITY_LOWEST <= p && p <= STHREADS_PRIORITY_HIGHEST;
}

```

```

/*-----*/
/* Print error message to string */
/*-----*/

```

```

void SthreadsWriteErrorMessage(int errorCode, char errorString[])

```

```

{
    switch (errorCode) {
    case STHEADS_ERROR_NONE:
        sprintf(errorString,
            "no error");
        break;
    case STHEADS_ERROR_INPUTVALUE:
        sprintf(errorString,
            "input value precondition violation");
        break;
    case STHEADS_ERROR_MEMORYALLOC:
        sprintf(errorString,
            "memory allocation failure");
        break;
    case STHEADS_ERROR_THREADCREATE:
        sprintf(errorString,
            "system thread creation failure");
        break;
    case STHEADS_ERROR_SYNC_CREATE:
        sprintf(errorString,
            "system synchronization creation failure");
        break;
    case STHEADS_ERROR_INITIALIZED:
        sprintf(errorString,
            "initialization on previously initialized object");
        break;
    case STHEADS_ERROR_UNINITIALIZED:
        sprintf(errorString,
            "operation on uninitialized object");
        break;
    case STHEADS_ERROR_FINALIZED:
        sprintf(errorString,
            "operation on finalized object");
        break;
    case STHEADS_ERROR_INUSE:
        sprintf(errorString,
            "finalization/reset on in-use object");
        break;
    case STHEADS_ERROR_LOCKNOTHELD:
        sprintf(errorString,
            "release on lock not held");
        break;
    case STHEADS_ERROR_COUNTEROVERFLOW:
        sprintf(errorString,
            "counter overflow");
        break;
    case STHEADS_ERROR_UNSPECIFIED:
        sprintf(errorString,
            "unspecified error");
        break;
    default:
        sprintf(errorString,
            ">>>> unknown error code <<<<");
        break;
    }
}

```

```

/*-----*/
/* Default error handler function: */
/* displays error message and terminate normal program execution. */
/*-----*/

```

```

static void DefaultErrorHandler(int errorCode)

```

```

{
    char errorString[STHEADS_ERROR_STRING_MAX];

    if (errorCode != STHEADS_ERROR_NONE) {

```

```

        SthreadsWriteErrorMessage(errorCode, errorString);
        fprintf(stderr, "\n%s\n", errorString);
        exit(EXIT_FAILURE);
    }
}

/*-----*/
/* Error handler function. */
/*-----*/

static void (*errorHandlerFunction)(int errorCode) = DefaultErrorHandler;

/*-----*/
/* Handle errors. */
/*-----*/

#define UNLOCKED 0
#define LOCKED 1

static LONG lock = UNLOCKED;

void SthreadsErrorHandler(int errorCode)
{
    while (InterlockedExchange((LPLONG) &lock, LOCKED) != UNLOCKED);
    (*errorHandlerFunction)(errorCode);
    InterlockedExchange((LPLONG) &lock, UNLOCKED);
}

#undef UNLOCKED
#undef LOCKED

/*-----*/
/* Set error handler function. */
/*-----*/

int SthreadsSetErrorHandler(void (*errorHandler)(int errorCode))
{
    if (errorHandler == NULL)
        errorHandlerFunction = DefaultErrorHandler;
    else
        errorHandlerFunction = errorHandler;
    return STHREADS_ERROR_NONE;
}

/*-----*/
/* Control the processors used by program execution. */
/*-----*/

int SthreadsGetSystemProcessors(int processor[])
{
    DWORD processAffinity, systemAffinity, processorBit;
    int p;

    require(STHEADS_PROCESSORS_MAX == 32);
    GetProcessAffinityMask(
        GetCurrentProcess(),
        (LPDWORD) &processAffinity, (LPDWORD) &systemAffinity);

    CHECKINPUTVALUE(processor != NULL);

    processorBit = (DWORD) 1;
    for (p = 0; p < STHREADS_PROCESSORS_MAX; p++) {
        if (systemAffinity & processorBit)
            processor[p] = STHREADS_PROCESSOR_YES;
        else
            processor[p] = STHREADS_PROCESSOR_NO;
        processorBit = processorBit << 1;
    }

    return STHREADS_ERROR_NONE;
}

/*-----*/

```

```

int SthreadsSetProgramProcessors(int processor[])
{
    DWORD processAffinity, systemAffinity, processorBit;
    int p;

    require(STHREADS_PROCESSORS_MAX == 32);
    GetProcessAffinityMask(
        GetCurrentProcess(),
        (LPDWORD) &processAffinity, (LPDWORD) &systemAffinity);

    CHECKINPUTVALUE(processor != NULL);
    processorBit = (DWORD) 1;
    for (p = 0; p < STHREADS_PROCESSORS_MAX; p++) {
        CHECKINPUTVALUE(ValidProcessorStatus(processor[p]));
        if (processor[p] == STHREADS_PROCESSOR_YES)
            CHECKINPUTVALUE(systemAffinity & processorBit);
        processorBit = processorBit << 1;
    }
    for (p = 0; p < STHREADS_PROCESSORS_MAX; p++)
        if (processor[p] == STHREADS_PROCESSOR_YES) break;
    CHECKINPUTVALUE(p < STHREADS_PROCESSORS_MAX);

    processAffinity = (DWORD) 0;
    processorBit = (DWORD) 1;
    for (p = 0; p < STHREADS_PROCESSORS_MAX; p++) {
        if (processor[p] == STHREADS_PROCESSOR_YES)
            processAffinity = processAffinity | processorBit;
        processorBit = processorBit << 1;
    }
    SetProcessAffinityMask(GetCurrentProcess(), processAffinity);
    SetThreadAffinityMask(GetCurrentThread(), processAffinity);

    return STHREADS_ERROR_NONE;
}

/*-----*/

int SthreadsGetProgramProcessors(int processor[])
{
    DWORD processAffinity, systemAffinity, processorBit;
    int p;

    require(STHREADS_PROCESSORS_MAX == 32);
    GetProcessAffinityMask(
        GetCurrentProcess(),
        (LPDWORD) &processAffinity, (LPDWORD) &systemAffinity);

    CHECKINPUTVALUE(processor != NULL);

    processorBit = (DWORD) 1;
    for (p = 0; p < STHREADS_PROCESSORS_MAX; p++) {
        if (processAffinity & processorBit)
            processor[p] = STHREADS_PROCESSOR_YES;
        else
            processor[p] = STHREADS_PROCESSOR_NO;
        processorBit = processorBit << 1;
    }

    return STHREADS_ERROR_NONE;
}

/*-----*/

int SthreadsSetThreadProcessors(int processor[])
{
    DWORD threadAffinity, processAffinity, systemAffinity, processorBit;
    int p;

    require(STHREADS_PROCESSORS_MAX == 32);
    GetProcessAffinityMask(
        GetCurrentProcess(),
        (LPDWORD) &processAffinity, (LPDWORD) &systemAffinity);

```

```

CHECKINPUTVALUE(processor != NULL);
processorBit = (DWORD) 1;
for (p = 0; p < STHREADS_PROCESSORS_MAX; p++) {
    CHECKINPUTVALUE(ValidProcessorStatus(processor[p]));
    if (processor[p] == STHREADS_PROCESSOR_YES)
        CHECKINPUTVALUE(processAffinity & processorBit);
    processorBit = processorBit << 1;
}
for (p = 0; p < STHREADS_PROCESSORS_MAX; p++)
    if (processor[p] == STHREADS_PROCESSOR_YES) break;
CHECKINPUTVALUE(p < STHREADS_PROCESSORS_MAX);

threadAffinity = (DWORD) 0;
processorBit = (DWORD) 1;
for (p = 0; p < STHREADS_PROCESSORS_MAX; p++) {
    if (processor[p] == STHREADS_PROCESSOR_YES)
        threadAffinity = threadAffinity | processorBit;
    processorBit = processorBit << 1;
}
SetThreadAffinityMask(GetCurrentThread(), threadAffinity);

return STHREADS_ERROR_NONE;
}

/*-----*/

int SthreadsGetNumSystemProcessors(int *numProcessors)
{
    DWORD processAffinity, systemAffinity, processorBit;
    int p, count;

    require(STHEADS_PROCESSORS_MAX == 32);
    GetProcessAffinityMask(
        GetCurrentProcess(),
        (LPDWORD) &processAffinity, (LPDWORD) &systemAffinity);

    CHECKINPUTVALUE(numProcessors != NULL);

    count = 0;
    processorBit = (DWORD) 1;
    for (p = 0; p < STHREADS_PROCESSORS_MAX; p++) {
        if (systemAffinity & processorBit)
            count = count + 1;
        processorBit = processorBit << 1;
    }
    *numProcessors = count;

    return STHREADS_ERROR_NONE;
}

/*-----*/

int SthreadsSetNumProgramProcessors(int numProcessors)
{
    DWORD processAffinity, systemAffinity, processorBit;
    int p, numSystemProcessors;

    require(STHEADS_PROCESSORS_MAX == 32);
    GetProcessAffinityMask(
        GetCurrentProcess(),
        (LPDWORD) &processAffinity, (LPDWORD) &systemAffinity);

    CHECKINPUTVALUE(numProcessors >= 1);
    numSystemProcessors = 0;
    processorBit = (DWORD) 1;
    for (p = 0; p < STHREADS_PROCESSORS_MAX; p++) {
        if (systemAffinity & processorBit)
            numSystemProcessors = numSystemProcessors + 1;
        processorBit = processorBit << 1;
    }
    CHECKINPUTVALUE(numProcessors <= numSystemProcessors);

    processAffinity = (DWORD) 0;
    processorBit = (DWORD) 1;

```

```

    for (p = 0; p < STTHREADS_PROCESSORS_MAX && numProcessors > 0; p++) {
        if (systemAffinity & processorBit) {
            processAffinity = processAffinity | processorBit;
            numProcessors = numProcessors - 1;
        }
        processorBit = processorBit << 1;
    }
    believe(numProcessors == 0);
    SetProcessAffinityMask(GetCurrentProcess(), processAffinity);

    return STTHREADS_ERROR_NONE;
}

/*-----*/
/* Arguments for multithreaded block thread */
/*-----*/

typedef struct {
    int numStatements;
    void (**statement)(void *args);
    void *args;
    int first, last, step;
    int *counter;
    LPCRITICAL_SECTION counterLock;
    LPLONG threadCount;
    HANDLE threadsFinished;
} MTBargs;

/*-----*/
/* Simple multithreaded block thread */
/*-----*/

static void SMTBthread(MTBargs *args)
{
    BOOL returnOK;

    require(args != NULL);
    require(args->numStatements > 0);
    require(args->statement != NULL);
    require(0 <= args->first && args->first < args->numStatements);
    require(args->statement[args->first] != NULL);

    (*args->statement[args->first])(args->args);

    if (InterlockedDecrement(&args->threadCount) == 0) {
        returnOK = SetEvent(args->threadsFinished);
        check(returnOK);
    }
}

/*-----*/
/* Dynamic multithreaded block thread */
/*-----*/

static void DMTBthread(MTBargs *args)
{
    int s;
    bool finished;
    BOOL returnOK;

    require(args != NULL);
    require(args->numStatements > 0);
    require(args->statement != NULL);
    require(0 <= args->first && args->first < args->numStatements);
    require(args->counter != NULL);
    require(args->counterLock != NULL);

    s = args->first;
    while (true) {
        require(args->statement[s] != NULL);
        (*args->statement[s])(args->args);
        EnterCriticalSection(&args->counterLock);
        finished = (*args->counter == args->numStatements - 1);
        if (!finished) {

```



```

        *args->counter = *args->counter + 1;
        s = *args->counter;
    }
    LeaveCriticalSection(args->counterLock);
    if (finished) break;
}

if (InterlockedDecrement(args->threadCount) == 0) {
    returnOK = SetEvent(args->threadsFinished);
    check(returnOK);
}
}

/*-----*/
/* Blocked and interleaved multithreaded block thread */
/*-----*/

static void BMTBthread(MTBargs *args)
{
    int s;
    BOOL returnOK;

    require(args != NULL);
    require(args->numStatements > 0);
    require(args->statement != NULL);
    require(0 <= args->last && args->last < args->numStatements);
    require(0 <= args->first && args->first <= args->last);
    require(args->step > 0);
    require((args->last - args->first)%args->step == 0);

    s = args->first;
    while (true) {
        require(args->statement[s] != NULL);
        (*args->statement[s])(args->args);
        if (s == args->last) break;
        believe(args->last - s >= args->step);
        s = s + args->step;
    }

    if (InterlockedDecrement(args->threadCount) == 0) {
        returnOK = SetEvent(args->threadsFinished);
        check(returnOK);
    }
}

/*-----*/
/* Multithreaded block */
/*-----*/

int SthreadsBlock(
    int numStatements, void (*statement[]) (void *args), void *args,
    int mapping, int numThreads,
    int priority, unsigned int stackSize)
{
    HANDLE *thread;
    MTBargs *threadArgs;
    LONG threadCount;
    HANDLE threadsFinished;
    HANDLE parentThread;
    int parentPriority;
    void (*threadStart) (MTBargs *args);
    int s, t;
    DWORD threadID;
    int counter;
    CRITICAL_SECTION counterLock;
    int blockFirst, blockSize, blockRemainder;
    BOOL returnOK;
    DWORD returnCode;

    CHECKINPUTVALUE(numStatements >= 0);
    CHECKINPUTVALUE(statement != NULL);
    for (s = 0; s < numStatements; s++)
        CHECKINPUTVALUE(statement[s] != NULL);
    CHECKINPUTVALUE(ValidMapping(mapping));

```

```

if (mapping != STHREADS_MAPPING_SIMPLE)
    CHECKINPUTVALUE((numThreads > 0) ||
        (numThreads == 0 && numStatements == 0));
CHECKINPUTVALUE(
    ValidPriority(priority) || priority == STHREADS_PRIORITY_PARENT);
CHECKINPUTVALUE(ValidStackSize(stackSize));

if (numStatements == 0) return STHREADS_ERROR_NONE;

if (mapping == STHREADS_MAPPING_SIMPLE) numThreads = numStatements;
if (numThreads > numStatements) numThreads = numStatements;
if (numThreads == 1) mapping = STHREADS_MAPPING_BLOCKED;
if (numThreads == numStatements) mapping = STHREADS_MAPPING_SIMPLE;

CHECKMEMORYALLOC(numThreads <= INT_MAX/sizeof(HANDLE));
thread = (HANDLE *) malloc(numThreads*sizeof(HANDLE));
CHECKMEMORYALLOC(thread != NULL);
CHECKMEMORYALLOC(numThreads <= INT_MAX/sizeof(MTBargs));
threadArgs = (MTBargs *) malloc(numThreads*sizeof(MTBargs));
CHECKMEMORYALLOC(threadArgs != NULL);

parentThread = GetCurrentThread();
believe(parentThread != NULL);
parentPriority = GetThreadPriority(parentThread);
believe(parentPriority != THREAD_PRIORITY_ERROR_RETURN);
believe(ValidPriority(parentPriority));
if (priority != STHREADS_PRIORITY_PARENT) {
    returnOK = SetThreadPriority(parentThread, priority);
    believe(returnOK);
}

switch (mapping) {
case STHREADS_MAPPING_SIMPLE:
    threadStart = SMTBthread;
    break;
case STHREADS_MAPPING_DYNAMIC:
    counter = numThreads - 1;
    InitializeCriticalSection(&counterLock);
    threadStart = DMTBthread;
    break;
case STHREADS_MAPPING_BLOCKED:
    blockFirst = 0;
    blockSize = numStatements/numThreads;
    blockRemainder = numStatements%numThreads;
    threadStart = BIMTBthread;
    break;
case STHREADS_MAPPING_INTERLEAVED:
    blockSize = numStatements/numThreads;
    blockRemainder = numStatements%numThreads;
    threadStart = BIMTBthread;
    break;
default:
    assert(false);
}

threadCount = numThreads;
threadsFinished = CreateEvent(NULL, TRUE, FALSE, NULL);
CHECKSYNCCREATE(threadsFinished != NULL);
for (t = 0; t < numThreads; t++) {
    threadArgs[t].numStatements = numStatements;
    threadArgs[t].statement = statement;
    threadArgs[t].args = args;
    threadArgs[t].threadCount = (LPLONG) &threadCount;
    threadArgs[t].threadsFinished = threadsFinished;

    switch (mapping) {
    case STHREADS_MAPPING_SIMPLE:
        threadArgs[t].first = t;
        break;
    case STHREADS_MAPPING_DYNAMIC:
        threadArgs[t].first = t;
        threadArgs[t].counter = &counter;
        threadArgs[t].counterLock = &counterLock;
        break;

```

```

    case STTHREADS_MAPPING_BLOCKED:
        threadArgs[t].first = blockFirst;
        threadArgs[t].last = blockFirst + (blockSize - 1);
        threadArgs[t].step = 1;
        if (blockRemainder > 0) {
            threadArgs[t].last = threadArgs[t].last + 1;
            blockRemainder = blockRemainder - 1;
        }
        blockFirst = threadArgs[t].last + 1;
        break;
    case STTHREADS_MAPPING_INTERLEAVED:
        threadArgs[t].first = t;
        threadArgs[t].last = blockSize*numThreads + t;
        threadArgs[t].step = numThreads;
        if (blockRemainder == 0)
            threadArgs[t].last = threadArgs[t].last - numThreads;
        else
            blockRemainder = blockRemainder - 1;
        break;
    default:
        believe(false);
}

thread[t] = CreateThread(NULL, stackSize,
    (LPTHREAD_START_ROUTINE) threadStart,
    (LPVOID) &threadArgs[t], CREATE_SUSPENDED, &threadID);
CHECKTHREADCREATE(thread[t] != NULL);
if (priority == STTHREADS_PRIORITY_PARENT)
    returnOK = SetThreadPriority(thread[t], parentPriority);
else
    returnOK = SetThreadPriority(thread[t], priority);
CHECKTHREADCREATE(returnOK);
returnCode = ResumeThread(thread[t]);
CHECKTHREADCREATE(returnCode == 1);
}

if (priority != STTHREADS_PRIORITY_PARENT) {
    returnOK = SetThreadPriority(parentThread, parentPriority);
    believe(returnOK);
}
returnCode = WaitForSingleObject(threadsFinished, INFINITE);
CHECKOTHER(returnCode != WAIT_FAILED);
returnOK = CloseHandle(threadsFinished);
CHECKOTHER(returnOK == TRUE);
for (t = 0; t < numThreads; t++) {
    returnOK = CloseHandle(thread[t]);
    CHECKOTHER(returnOK == TRUE);
}
if (mapping == STTHREADS_MAPPING_DYNAMIC)
    DeleteCriticalSection(&counterLock);
free(thread);
free(threadArgs);

return STTHREADS_ERROR_NONE;
}

```

```

/*-----*/
/* Is regular for loop range infinite? */
/*-----*/

```

```

static bool InfiniteRange(int initial, int condition, int bound, int step)
{
    require(ValidCondition(condition));

    switch (condition) {
    case STTHREADS_CONDITION_LT:
        return initial < bound && step <= 0;
    case STTHREADS_CONDITION_LE:
        return initial <= bound && step <= 0;
    case STTHREADS_CONDITION_GT:
        return initial > bound && step >= 0;
    case STTHREADS_CONDITION_GE:
        return initial >= bound && step >= 0;
    default:

```

```

        believe(false);
        return false; /* This return should never be executed. */
    }
}

/*-----*/
/* Is regular for loop range null? */
/*-----*/

static bool NullRange(int initial, int condition, int bound, int step)
{
    require(ValidCondition(condition));

    switch (condition) {
    case STTHREADS_CONDITION_LT:
        return initial >= bound;
    case STTHREADS_CONDITION_LE:
        return initial > bound;
    case STTHREADS_CONDITION_GT:
        return initial <= bound;
    case STTHREADS_CONDITION_GE:
        return initial < bound;
    default:
        believe(false);
        return false; /* This return should never be executed. */
    }
}

/*-----*/
/* Arithmetic operations on signed and unsigned integers */
/*-----*/

static unsigned int DIFF(int high, int low)
{
    require(low <= high);

    return (unsigned int) (high - low);
}

/*-----*/

static int ADD(int base, unsigned int offset)
{
    require(offset <= DIFF(INT_MAX, base));

    return base + (int) offset;
}

/*-----*/

static int SUBTRACT(int base, unsigned int offset)
{
    require(offset <= DIFF(base, INT_MIN));

    return base - (int) offset;
}

/*-----*/
/* Split range 0 .. rangeLast into chunks numbered 0 .. chunkLast with */
/* chunks. Return the first and last indices of chunk c. */
/*-----*/

static void SPLIT(
    unsigned int rangeLast, unsigned int chunkLast, unsigned int c,
    unsigned int *first, unsigned int *last)
{
    unsigned int smallerChunkSize;
    unsigned int numLargerChunks;

    require(chunkLast <= rangeLast);
    require(c <= chunkLast);
    require(first != NULL && last != NULL);

    if (chunkLast == 0) {

```

```

    *first = 0;
    *last = rangeLast;
} else if (chunkLast == rangeLast) {
    *first = c;
    *last = c;
} else {
    smallerChunkSize = (rangeLast - chunkLast)/(chunkLast + 1) + 1;
    numLargerChunks = (rangeLast - chunkLast)%(chunkLast + 1);
    *first = c*smallerChunkSize + MIN(c, numLargerChunks);
    *last = *first + (smallerChunkSize - 1);
    if (c < numLargerChunks) *last = *last + 1;
}
}

/*-----*/
/* Last iteration number in regular for loop range */
/* (iterations numbered 0, 1, 2, ...) */
/*-----*/

static unsigned int LAST_ITERATION_NUM(
    int initial, int condition, int bound, int step)
{
    require(ValidCondition(condition));
    require(!InfiniteRange(initial, condition, bound, step));
    require(!NullRange(initial, condition, bound, step));

    switch (condition) {
    case STHTREADS_CONDITION_LT:
        believe(initial < bound && step > 0);
        return DIFF(bound - 1, initial)/((unsigned int) step);
    case STHTREADS_CONDITION_LE:
        believe(initial <= bound && step > 0);
        return DIFF(bound, initial)/((unsigned int) step);
    case STHTREADS_CONDITION_GT:
        believe(initial > bound && step < 0);
        return DIFF(initial, bound + 1)/((unsigned int) -step);
    case STHTREADS_CONDITION_GE:
        believe(initial >= bound && step < 0);
        return DIFF(initial, bound)/((unsigned int) -step);
    default:
        assert(false);
        return false; /* This return should never be executed. */
    }
}

/*-----*/
/* Last chunk number in regular for loop range (chunks numbered 0, 1, 2, ...) */
/*-----*/

static unsigned int LAST_CHUNK_NUM(
    int initial, int condition, int bound, int step, int chunkSize)
{
    require(ValidCondition(condition));
    require(!InfiniteRange(initial, condition, bound, step));
    require(!NullRange(initial, condition, bound, step));
    require(chunkSize >= 1);

    return LAST_ITERATION_NUM(initial, condition, bound, step)/
        ((unsigned int) chunkSize);
}

/*-----*/
/* Control value on ith iteration of regular for loop range (i = 0, 1, 2, ...) */
/*-----*/

static int ControlValue(unsigned int i, int initial, int step)
{
    require(step != 0);

    if (step > 0)
        return ADD(initial, i*((unsigned int) step));
    else
        return SUBTRACT(initial, i*((unsigned int) -step));
}

```

```

/*-----*/
/* Does control value lie inside regular for loop range? */
/*-----*/

static bool InRange(
    int controlValue, int initial, int condition, int bound, int step)
{
    require(ValidCondition(condition));
    require(!InfiniteRange(initial, condition, bound, step));
    require(!NullRange(initial, condition, bound, step));

    switch (condition) {
    case STHREADS_CONDITION_LT:
        believe(step > 0);
        return initial <= controlValue && controlValue < bound;
    case STHREADS_CONDITION_LE:
        believe(step > 0);
        return initial <= controlValue && controlValue <= bound;
    case STHREADS_CONDITION_GT:
        believe(step < 0);
        return initial >= controlValue && controlValue > bound;
    case STHREADS_CONDITION_GE:
        believe(step < 0);
        return initial >= controlValue && controlValue >= bound;
    default:
        believe(false);
        return false; /* This return should never be executed. */
    }
}

/*-----*/
/* Execute cth chunk of regular for loop range (c = 0, 1, 2, ...) */
/*-----*/

static void ExecuteChunk(
    int initial, int condition, int bound, int step, int chunkSize,
    unsigned int c, void (*chunk)(int, int, int, void *), void *args)
{
    unsigned int iFirst, iLast;
    int chunkInitial, chunkLast, chunkBound;

    require(ValidCondition(condition));
    require(!InfiniteRange(initial, condition, bound, step));
    require(!NullRange(initial, condition, bound, step));
    require(chunkSize >= 1);
    require(c <= LAST_CHUNK_NUM(initial, condition, bound, step, chunkSize));
    require(chunk != NULL);

    SPLIT(
        LAST_ITERATION_NUM(initial, condition, bound, step),
        LAST_CHUNK_NUM(initial, condition, bound, step, chunkSize), c,
        &iFirst, &iLast);
    believe(0 <= iFirst);
    believe(iFirst <= iLast);
    believe(iLast <= LAST_ITERATION_NUM(initial, condition, bound, step));
    chunkInitial = ControlValue(iFirst, initial, step);
    believe(InRange(chunkInitial, initial, condition, bound, step));
    chunkLast = ControlValue(iLast, initial, step);
    believe(InRange(chunkLast, initial, condition, bound, step));
    switch (condition) {
    case STHREADS_CONDITION_LT:
        chunkBound = chunkLast + 1;
        break;
    case STHREADS_CONDITION_LE:
        chunkBound = chunkLast;
        break;
    case STHREADS_CONDITION_GT:
        chunkBound = chunkLast - 1;
        break;
    case STHREADS_CONDITION_GE:
        chunkBound = chunkLast;
        break;
    default:

```

```

        believe(false);
    }
    (*chunk)(chunkInitial, chunkBound, step, args);
}

/*-----*/
/* Arguments for multithreaded regular for loop thread */
/*-----*/

typedef struct {
    void (*chunk)(int initial, int bound, int step, void *args);
    void *args;
    int initial, condition, bound, step;
    int chunkSize;
    unsigned int chunkFirst, chunkLast, chunkStep;
    unsigned int *counter;
    LPCRITICAL_SECTION counterLock;
    LPLONG threadCount;
    HANDLE threadsFinished;
} MTRFLargs;

/*-----*/
/* Simple multithreaded regular for loop thread */
/*-----*/

static void SMTRFLthread(MTRFLargs *args)
{
    BOOL returnOK;

    require(args != NULL);
    require(args->chunk != NULL);
    require(ValidCondition(args->condition));
    require(!InfiniteRange(
        args->initial, args->condition, args->bound, args->step));
    require(!NullRange(
        args->initial, args->condition, args->bound, args->step));
    require(args->chunkSize >= 1);
    require(args->chunkFirst <= LAST_CHUNK_NUM(
        args->initial, args->condition, args->bound, args->step,
        args->chunkSize));

    ExecuteChunk(
        args->initial, args->condition, args->bound, args->step,
        args->chunkSize, args->chunkFirst, args->chunk, args->args);

    if (InterlockedDecrement(&args->threadCount) == 0) {
        returnOK = SetEvent(args->threadsFinished);
        check(returnOK);
    }
}

/*-----*/
/* Dynamic multithreaded regular for loop thread */
/*-----*/

static void DMTRFLthread(MTRFLargs *args)
{
    unsigned int c, last_c;
    bool finished;
    BOOL returnOK;

    require(args != NULL);
    require(args->chunk != NULL);
    require(ValidCondition(args->condition));
    require(!InfiniteRange(
        args->initial, args->condition, args->bound, args->step));
    require(!NullRange(
        args->initial, args->condition, args->bound, args->step));
    require(args->chunkSize >= 1);
    require(args->chunkFirst <= LAST_CHUNK_NUM(
        args->initial, args->condition, args->bound, args->step,
        args->chunkSize));
    require(args->counter != NULL);
    require(args->counterLock != NULL);

```

```

c = args->chunkFirst;
last_c = LAST_CHUNK_NUM(
    args->initial, args->condition, args->bound, args->step,
    args->chunkSize);
while (true) {
    ExecuteChunk(
        args->initial, args->condition, args->bound, args->step,
        args->chunkSize, c, args->chunk, args->args);
    EnterCriticalSection(args->counterLock);
    finished = (*args->counter == last_c);
    if (!finished) {
        *args->counter = *args->counter + 1;
        c = *args->counter;
    }
    LeaveCriticalSection(args->counterLock);
    if (finished) break;
}

if (InterlockedDecrement(args->threadCount) == 0) {
    returnOK = SetEvent(args->threadsFinished);
    check(returnOK);
}
}

/*-----*/
/* Blocked and interleaved multithreaded regular for loop thread */
/*-----*/

static void BIMTRFLthread(MTRFLargs *args)
{
    unsigned int c;
    BOOL returnOK;

    require(args != NULL);
    require(args->chunk != NULL);
    require(ValidCondition(args->condition));
    require(!InfiniteRange(
        args->initial, args->condition, args->bound, args->step));
    require(!NullRange(
        args->initial, args->condition, args->bound, args->step));
    require(args->chunkSize >= 1);
    require(args->chunkFirst <= args->chunkLast);
    require(args->chunkLast <= LAST_CHUNK_NUM(
        args->initial, args->condition, args->bound, args->step,
        args->chunkSize));
    require((args->chunkLast - args->chunkFirst)%args->chunkStep == 0);

    c = args->chunkFirst;
    while (true) {
        ExecuteChunk(
            args->initial, args->condition, args->bound, args->step,
            args->chunkSize, c, args->chunk, args->args);
        if (c == args->chunkLast) break;
        believe(args->chunkLast - c >= args->chunkStep);
        c = c + args->chunkStep;
    }

    if (InterlockedDecrement(args->threadCount) == 0) {
        returnOK = SetEvent(args->threadsFinished);
        check(returnOK);
    }
}

/*-----*/
/* Multithreaded regular for loop */
/*-----*/

int SthreadsRegularForLoop(
    void (*chunk)(int initial, int bound, int step, void *args), void *args,
    int initial, int condition, int bound, int step,
    int chunkSize, int mapping, int numThreads,
    int priority, unsigned int stackSize)
{

```



```

unsigned int lastChunkNum;
HANDLE *thread;
MTRFLargs *threadArgs;
LONG threadCount;
HANDLE threadsFinished;
HANDLE parentThread;
int parentPriority;
void (*thread_start)(MTRFLargs *args);
int t;
DWORD threadID;
int counter;
CRITICAL_SECTION counterLock;
unsigned int blockFirst, blockSize, blockRemainder;
BOOL returnOK;
DWORD returnCode;

CHECKINPUTVALUE(chunk != NULL);
CHECKINPUTVALUE(ValidCondition(condition));
CHECKINPUTVALUE(!InfiniteRange(initial, condition, bound, step));
CHECKINPUTVALUE((chunkSize > 0) ||
    (chunkSize == 0 &&
        NullRange(initial, condition, bound, step)));
CHECKINPUTVALUE(ValidMapping(mapping));
if (mapping != STHREADS_MAPPING_SIMPLE)
    CHECKINPUTVALUE((numThreads > 0) ||
        (numThreads == 0 &&
            NullRange(initial, condition, bound, step)));
CHECKINPUTVALUE(
    ValidPriority(priority) || priority == STHREADS_PRIORITY_PARENT);
CHECKINPUTVALUE(ValidStackSize(stackSize));

if (NullRange(initial, condition, bound, step))
    return STHREADS_ERROR_NONE;

lastChunkNum = LAST_CHUNK_NUM(
    initial, condition, bound, step, chunkSize);
CHECKMEMORYALLOC(!(mapping == STHREADS_MAPPING_SIMPLE &&
    lastChunkNum >= INT_MAX));

if (mapping == STHREADS_MAPPING_SIMPLE)
    numThreads = (int) (lastChunkNum + 1);
if ((unsigned int) (numThreads - 1) > lastChunkNum)
    numThreads = (int) (lastChunkNum + 1);
if (numThreads == 1)
    mapping = STHREADS_MAPPING_INTERLEAVED;
if ((unsigned int) (numThreads - 1) == lastChunkNum)
    mapping = STHREADS_MAPPING_SIMPLE;

CHECKMEMORYALLOC(numThreads <= INT_MAX/sizeof(HANDLE));
thread = (HANDLE *) malloc(numThreads*sizeof(HANDLE));
CHECKMEMORYALLOC(thread != NULL);
CHECKMEMORYALLOC(numThreads <= INT_MAX/sizeof(MTRFLargs));
threadArgs = (MTRFLargs *) malloc(numThreads*sizeof(MTRFLargs));
CHECKMEMORYALLOC(threadArgs != NULL);

parentThread = GetCurrentThread();
believe(parentThread != NULL);
parentPriority = GetThreadPriority(parentThread);
believe(parentPriority != THREAD_PRIORITY_ERROR_RETURN);
believe(ValidPriority(parentPriority));
if (priority != STHREADS_PRIORITY_PARENT) {
    returnOK= SetThreadPriority(parentThread, priority);
    believe(returnOK);
}

switch (mapping) {
case STHREADS_MAPPING_SIMPLE:
    thread_start = SMTRFLthread;
    break;
case STHREADS_MAPPING_DYNAMIC:
    counter = numThreads - 1;
    InitializeCriticalSection(&counterLock);
    thread_start = DMTRFLthread;
    break;

```

```

case STHEADS_MAPPING_BLOCKED:
    blockFirst = 0;
    blockSize =
        (lastChunkNum - (((unsigned int) numThreads) - 1)) /
        ((unsigned int) numThreads) + 1;
    blockRemainder =
        (lastChunkNum - (((unsigned int) numThreads) - 1)) %
        ((unsigned int) numThreads);
    thread_start = BIMTRFLthread;
    break;
case STHEADS_MAPPING_INTERLEAVED:
    blockSize =
        (lastChunkNum - (((unsigned int) numThreads) - 1)) /
        ((unsigned int) numThreads) + 1;
    blockRemainder =
        (lastChunkNum - (((unsigned int) numThreads) - 1)) %
        ((unsigned int) numThreads);
    thread_start = BIMTRFLthread;
    break;
default:
    assert(false);
}

threadCount = numThreads;
threadsFinished = CreateEvent(NULL, TRUE, FALSE, NULL);
CHECKSYNCCREATE(threadsFinished != NULL);
for (t = 0; t < numThreads; t++) {
    threadArgs[t].chunk = chunk;
    threadArgs[t].args = args;
    threadArgs[t].initial = initial;
    threadArgs[t].condition = condition;
    threadArgs[t].bound = bound;
    threadArgs[t].step = step;
    threadArgs[t].chunkSize = chunkSize;
    threadArgs[t].threadCount = (LPLONG) &threadCount;
    threadArgs[t].threadsFinished = threadsFinished;

    switch (mapping) {
    case STHEADS_MAPPING_SIMPLE:
        threadArgs[t].chunkFirst = t;
        break;
    case STHEADS_MAPPING_DYNAMIC:
        threadArgs[t].chunkFirst = t;
        threadArgs[t].counter = &counter;
        threadArgs[t].counterLock = &counterLock;
        break;
    case STHEADS_MAPPING_BLOCKED:
        threadArgs[t].chunkFirst = blockFirst;
        threadArgs[t].chunkLast = blockFirst + (blockSize - 1);
        threadArgs[t].chunkStep = 1;
        if (blockRemainder > 0) {
            threadArgs[t].chunkLast = threadArgs[t].chunkLast + 1;
            blockRemainder = blockRemainder - 1;
        }
        blockFirst = threadArgs[t].chunkLast + 1;
        break;
    case STHEADS_MAPPING_INTERLEAVED:
        threadArgs[t].chunkFirst = t;
        threadArgs[t].chunkLast =
            blockSize * ((unsigned int) numThreads) + t;
        threadArgs[t].chunkStep =
            (unsigned int) numThreads;
        if (blockRemainder == 0)
            threadArgs[t].chunkLast =
                threadArgs[t].chunkLast - ((unsigned int) numThreads);
        else
            blockRemainder = blockRemainder - 1;
        break;
    default:
        believe(false);
    }

    thread[t] = CreateThread(NULL, stackSize,
        (LPTHREAD_START_ROUTINE) thread_start,

```

```

        (LPVOID) &threadArgs[t], CREATE_SUSPENDED, &threadID);
CHECKTHREADCREATE(thread[t] != NULL);
if (priority == STHREADS_PRIORITY_PARENT)
    SetThreadPriority(thread[t], parentPriority);
else
    SetThreadPriority(thread[t], priority);
ResumeThread(thread[t]);
}

if (priority != STHREADS_PRIORITY_PARENT) {
    SetThreadPriority(parentThread, parentPriority);
    believe(returnOK);
}
returnCode = WaitForSingleObject(threadsFinished, INFINITE);
CHECKOTHER(returnCode != WAIT_FAILED);
returnOK = CloseHandle(threadsFinished);
CHECKOTHER(returnOK == TRUE);
for (t = 0; t < numThreads; t++) {
    returnOK = CloseHandle(thread[t]);
    CHECKOTHER(returnOK == TRUE);
}
if (mapping == STHREADS_MAPPING_DYNAMIC)
    DeleteCriticalSection(&counterLock);
free(thread);
free(threadArgs);

return STHREADS_ERROR_NONE;
}

/*-----*/
/*Multithreaded nested regular for loop (for future release?)*/
/*-----*/

int SthreadsNestedRegularForLoop(
    int nesting,
    void (*chunk)(int first[], int last[], int step[], void *args),
    void *args,
    int initial[], int condition[], int bound[], int step[],
    int chunkSize[], int mapping[], int numThreads[],
    int priority, unsigned int stackSize)
/*Arguments:*/
/* - nesting      : degree of nesting.*/
/* - chunk        : function to execute chunk of iterations of loop body.*/
/* - args         : pointer to arguments of loop body.*/
/* - initial      : initial value of control variable at each nesting level.*/
/* - condition    : condition between control variable and bound value*/
/*                 at each nesting level.*/
/* - bound        : bound value of control variable at each nesting level.*/
/* - step         : step value of control variable at each nesting level.*/
/* - chunkSize    : number of iterations per chunk at each nesting level.*/
/* - mapping      : mapping of chunks onto threads at each nesting level.*/
/* - numThreads   : number of threads at each nesting level.*/
/* - priority     : priority of threads.*/
/* - stackSize    : stack size of threads.*/
/* Returns:*/
/* - error code.*/
/* Requirements:*/
/* - nesting >= 1*/
/* - chunk != NULL &&*/
/*   chunk is a valid void (*)(int *, int *, int *, void *) function.*/
/* - initial != NULL &&*/
/*   initial is an array of at least nesting ints.*/
/* - condition != NULL &&*/
/*   condition is an array of at least nesting ints.*/
/* - forall (i = 0; i < nesting; i++) ValidCondition(condition[i]).*/
/* - bound != NULL &&*/
/*   bound is an array of at least nesting ints.*/
/* - step != NULL &&*/
/*   step is an array of at least nesting ints.*/
/* - forall (i = 0; i < nesting; i++)*/
/*     !InfiniteRange(initial[i], condition[i], bound[i], step[i]) ||*/
/*     exists (j = 0; j < i; j++)*/
/*       NullRange(initial[j], condition[j], bound[j], step[j]).*/
/* - forall (i = 0; i < nesting; i++)*/

```

```

/*      (chunkSize[i] > 0) ||
/*      (chunkSize[i] == 0 &&
/*      NullRange(initial[i], condition[i], bound[i], step[i])).
/* - forall (i = 0; i < nesting; i++) ValidMapping(mapping[i]).
/* - forall (i = 0; i < nesting; i++)
/*      mapping[i] != STHEADS_MAPPING_SIMPLE =>
/*      (numThreads[i] > 0) ||
/*      (numThreads[i] == 0 &&
/*      NullRange(initial[i], condition[i], bound[i], step[i])).
/* - ValidPriority(priority) || priority == STHEADS_PRIORITY_PARENT.
/* - ValidStackSize(stackSize).
{
    int i;

    CHECKINPUTVALUE(nesting >= 1);
    CHECKINPUTVALUE(chunk != NULL);
    CHECKINPUTVALUE(initial != NULL);
    CHECKINPUTVALUE(condition != NULL);
    for (i = 0; i < nesting; i++)
        CHECKINPUTVALUE(ValidCondition(condition[i]));
    CHECKINPUTVALUE(bound != NULL);
    CHECKINPUTVALUE(step != NULL);
    for (i = 0; i < nesting; i++) {
        if (NullRange(initial[i], condition[i], bound[i], step[i])) break;
        CHECKINPUTVALUE(
            !InfiniteRange(initial[i], condition[i], bound[i], step[i]));
    }
    for (i = 0; i < nesting; i++)
        CHECKINPUTVALUE((chunkSize[i] > 0) ||
            (chunkSize[i] == 0 &&
                NullRange(initial[i], condition[i], bound[i], step[i])));
    for (i = 0; i < nesting; i++)
        CHECKINPUTVALUE(ValidMapping(mapping[i]));
    for (i = 0; i < nesting; i++)
        if (mapping[i] != STHEADS_MAPPING_SIMPLE)
            CHECKINPUTVALUE(
                (numThreads[i] > 0) ||
                (numThreads[i] == 0 &&
                    NullRange(initial[i], condition[i], bound[i], step[i])));
    CHECKINPUTVALUE(
        ValidPriority(priority) || priority == STHEADS_PRIORITY_PARENT);
    CHECKINPUTVALUE(ValidStackSize(stackSize));

    return STHEADS_ERROR_NONE;
}

/*-----*/
/* Multithreaded general for loop (for future release?)
/*-----*/

int SthreadsGeneralForLoop(
    void (*body)(void *control, void *args),
    size_t controlSize, void *args,
    int (*test)(void *args), void (*increment)(void *args),
    void (*copy)(void *control, void *args),
    int mapping, int numThreads,
    int priority, unsigned int stackSize)
/* Arguments:
/* - body      : function to execute one iteration of loop body.
/* - controlSize : size (as returned by sizeof) of control variables.
/* - args      : pointer to arguments of loop.
/* - test      : function to test loop termination condition.
/* - increment  : function to increment control variables within arguments.
/* - copy      : function to copy control variables from arguments.
/* - mapping    : mapping of iterations onto threads.
/* - numThreads : number of threads.
/* - priority   : priority of threads.
/* - stackSize  : stack size of threads.
/* Returns:
/* - error code.
/* Requirements:
/* - body != NULL &&
/*   body is a valid void (*)(void *, void *) function.
/* - test != NULL &&

```

```

/* test is a valid int (*)(void *) function. */
/* - increment != NULL && */
/* increment is a valid void (*)(void *) function. */
/* - copy != NULL && */
/* copy is a valid void (*)(void *, void *) function. */
/* - mapping == STTHREADS_MAPPING_SIMPLE || */
/* mapping == STTHREADS_MAPPING_DYNAMIC. */
/* - mapping != STTHREADS_MAPPING_SIMPLE => */
/* (numThreads > 0) || (numThreads == 0 && !test(args)). */
/* - ValidPriority(priority) || priority == STTHREADS_PRIORITY_PARENT. */
/* - ValidStackSize(stackSize). */
{
    CHECKINPUTVALUE(body != NULL);
    CHECKINPUTVALUE(test != NULL);
    CHECKINPUTVALUE(increment != NULL);
    CHECKINPUTVALUE(copy != NULL);
    CHECKINPUTVALUE(mapping == STTHREADS_MAPPING_SIMPLE ||
        mapping == STTHREADS_MAPPING_DYNAMIC);
    if (mapping != STTHREADS_MAPPING_SIMPLE)
        CHECKINPUTVALUE((numThreads > 0) || (numThreads == 0 && !test(args)));
    CHECKINPUTVALUE(
        ValidPriority(priority) || priority == STTHREADS_PRIORITY_PARENT);
    CHECKINPUTVALUE(ValidStackSize(stackSize));

    return STTHREADS_ERROR_NONE;
}

/*-----*/
/* Synchronization object status constants */
/*-----*/
#define INITIALIZED 123456
#define FINALIZED 654321
/*-----*/
/* Flags */
/*-----*/
typedef struct {
    int initialized, finalized;
    LONG numWaiting;
    HANDLE signal;
} PrivateFlag;

#define PRIVATE(flagPtr) ((PrivateFlag *) (flagPtr))

/*-----*/
int SthreadsFlagInitialize(SthreadsFlag *flag)
{
    CHECKINPUTVALUE(flag != NULL);

    PRIVATE(flag)->initialized = INITIALIZED;
    PRIVATE(flag)->finalized = ~FINALIZED;
    PRIVATE(flag)->numWaiting = 0;
    PRIVATE(flag)->signal = CreateEvent(NULL, TRUE, FALSE, NULL);
    CHECKSYNCCREATE(PRIVATE(flag)->signal != NULL);

    return STTHREADS_ERROR_NONE;
}

/*-----*/
int SthreadsFlagFinalize(SthreadsFlag *flag)
{
    BOOL returnOK;

    CHECKINPUTVALUE(flag != NULL);
    CHECKUNINITIALIZED(PRIVATE(flag)->initialized == INITIALIZED);
    CHECKFINALIZED(PRIVATE(flag)->finalized == ~FINALIZED);
    CHECKINUSE(PRIVATE(flag)->numWaiting == 0);

    PRIVATE(flag)->finalized = FINALIZED;
    returnOK = CloseHandle(PRIVATE(flag)->signal);
}

```

```

    CHECKOTHER(returnOK == TRUE);

    return STHREADS_ERROR_NONE;
}

/*-----*/

int SthreadsFlagSet(SthreadsFlag *flag)
{
    BOOL returnOK;

    CHECKINPUTVALUE(flag != NULL);
    CHECKUNINITIALIZED(PRIVATE(flag)->initialized == INITIALIZED);
    CHECKFINALIZED(PRIVATE(flag)->finalized == ~FINALIZED);

    returnOK = SetEvent(PRIVATE(flag)->signal);
    CHECKOTHER(returnOK);

    return STHREADS_ERROR_NONE;
}

/*-----*/

int SthreadsFlagCheck(SthreadsFlag *flag)
{
    DWORD returnCode;

    CHECKINPUTVALUE(flag != NULL);
    CHECKUNINITIALIZED(PRIVATE(flag)->initialized == INITIALIZED);
    CHECKFINALIZED(PRIVATE(flag)->finalized == ~FINALIZED);

    InterlockedIncrement(&PRIVATE(flag)->numWaiting);
    returnCode = WaitForSingleObject(PRIVATE(flag)->signal, INFINITE);
    CHECKOTHER(returnCode != WAIT_FAILED);
    InterlockedDecrement(&PRIVATE(flag)->numWaiting);

    return STHREADS_ERROR_NONE;
}

/*-----*/

int SthreadsFlagReset(SthreadsFlag *flag)
{
    BOOL returnOK;

    CHECKINPUTVALUE(flag != NULL);
    CHECKUNINITIALIZED(PRIVATE(flag)->initialized == INITIALIZED);
    CHECKFINALIZED(PRIVATE(flag)->finalized == ~FINALIZED);
    CHECKINUSE(PRIVATE(flag)->numWaiting == 0);

    PRIVATE(flag)->numWaiting = 0;
    returnOK = ResetEvent(PRIVATE(flag)->signal);
    CHECKOTHER(returnOK);

    return STHREADS_ERROR_NONE;
}

/*-----*/

#undef PRIVATE

/*-----*/
/* Counters */
/*-----*/

typedef struct node *link;
typedef struct node {
    unsigned int value;
    int numWaiting;
    HANDLE signal;
    link next;
} node;

typedef struct {

```

```

    int initialized, finalized;
    unsigned int count;
    link waitingList;
    CRITICAL_SECTION lock;
} PrivateCounter;

#define PRIVATE(counterPtr) ((PrivateCounter *) (counterPtr))

/*-----*/

int SthreadsCounterInitialize(SthreadsCounter *counter)
{
    link startSentinel, endSentinel;

    CHECKINPUTVALUE(counter != NULL);

    PRIVATE(counter)->initialized = INITIALIZED;
    PRIVATE(counter)->finalized = ~FINALIZED;
    PRIVATE(counter)->count = 0;
    startSentinel = (link) malloc(sizeof(node));
    CHECKMEMORYALLOC(startSentinel != NULL);
    endSentinel = (link) malloc(sizeof(node));
    CHECKMEMORYALLOC(endSentinel != NULL);
    startSentinel->signal = NULL;
    startSentinel->next = endSentinel;
    startSentinel->numWaiting = 0;
    endSentinel->signal = NULL;
    endSentinel->next = NULL;
    endSentinel->numWaiting = 0;
    PRIVATE(counter)->waitingList = startSentinel;
    InitializeCriticalSection((LPCRITICAL_SECTION) &PRIVATE(counter)->lock);

    return STHREADS_ERROR_NONE;
}

/*-----*/

int SthreadsCounterFinalize(SthreadsCounter *counter)
{
    link p, next;
    BOOL returnOK;

    CHECKINPUTVALUE(counter != NULL);
    CHECKUNINITIALIZED(PRIVATE(counter)->initialized == INITIALIZED);
    CHECKFINALIZED(PRIVATE(counter)->finalized == ~FINALIZED);
    CHECKINUSE(PRIVATE(counter)->waitingList->next->next == NULL);

    PRIVATE(counter)->finalized = FINALIZED;
    p = PRIVATE(counter)->waitingList;
    next = p->next;
    free(p);
    p = next;
    while (p->next != NULL) {
        returnOK = CloseHandle(p->signal);
        CHECKOTHER(returnOK == TRUE);
        next = p->next;
        free(p);
        p = next;
    }
    free(p);
    DeleteCriticalSection((LPCRITICAL_SECTION) &PRIVATE(counter)->lock);

    return STHREADS_ERROR_NONE;
}

/*-----*/

int SthreadsCounterIncrement(SthreadsCounter *counter, unsigned int amount)
{
    link start, p;
    BOOL returnOK;

    CHECKINPUTVALUE(counter != NULL);
    CHECKUNINITIALIZED(PRIVATE(counter)->initialized == INITIALIZED);

```

```

CHECKFINALIZED(PRIVATE(counter)->finalized == ~FINALIZED);
CHECKCOUNTEROVERFLOW(PRIVATE(counter)->count <= UINT_MAX - amount);

EnterCriticalSection((LPCRITICAL_SECTION) &PRIVATE(counter)->lock);
PRIVATE(counter)->count = PRIVATE(counter)->count + amount;
start = PRIVATE(counter)->waitingList;
p = start->next;
while (p->next != NULL && p->value <= PRIVATE(counter)->count) {
    returnOK = SetEvent(p->signal);
    CHECKOTHER(returnOK);
    start->next = p->next;
    p = start->next;
}
LeaveCriticalSection((LPCRITICAL_SECTION) &PRIVATE(counter)->lock);

return STHREADS_ERROR_NONE;
}

/*-----*/

int SthreadsCounterCheck(SthreadsCounter *counter, unsigned int value)
{
    link prev, p;
    link waitingNode;
    BOOL returnOK;
    DWORD returnCode;

    CHECKINPUTVALUE(counter != NULL);
    CHECKUNINITIALIZED(PRIVATE(counter)->initialized == INITIALIZED);
    CHECKFINALIZED(PRIVATE(counter)->finalized == ~FINALIZED);

    EnterCriticalSection((LPCRITICAL_SECTION) &PRIVATE(counter)->lock);
    if (PRIVATE(counter)->count >= value)
        LeaveCriticalSection((LPCRITICAL_SECTION) &PRIVATE(counter)->lock);
    else {
        prev = PRIVATE(counter)->waitingList;
        p = prev->next;
        while (p->next != NULL && p->value < value) {
            prev = p;
            p = p->next;
        }
        if (p->value == value) {
            waitingNode = p;
            waitingNode->numWaiting = waitingNode->numWaiting + 1;
        } else {
            waitingNode = (link) malloc(sizeof(node));
            waitingNode->value = value;
            waitingNode->signal = CreateEvent(NULL, TRUE, FALSE, NULL);
            waitingNode->next = p;
            waitingNode->numWaiting = 1;
            prev->next = waitingNode;
        }
        LeaveCriticalSection((LPCRITICAL_SECTION) &PRIVATE(counter)->lock);
        returnCode = WaitForSingleObject(waitingNode->signal, INFINITE);
        CHECKOTHER(returnCode != WAIT_FAILED);
        EnterCriticalSection((LPCRITICAL_SECTION) &PRIVATE(counter)->lock);
        waitingNode->numWaiting = waitingNode->numWaiting - 1;
        if (waitingNode->numWaiting == 0) {
            returnOK = CloseHandle(waitingNode->signal);
            CHECKOTHER(returnOK == TRUE);
            free(waitingNode);
        }
        LeaveCriticalSection((LPCRITICAL_SECTION) &PRIVATE(counter)->lock);
    }

    return STHREADS_ERROR_NONE;
}

/*-----*/

int SthreadsCounterReset(SthreadsCounter *counter)
{
    link p, q;
    BOOL returnOK;

```



```

CHECKINPUTVALUE(counter != NULL);
CHECKUNINITIALIZED(PRIVATE(counter)->initialized == INITIALIZED);
CHECKFINALIZED(PRIVATE(counter)->finalized == ~FINALIZED);
CHECKINUSE(PRIVATE(counter)->waitingList->next->next == NULL);

PRIVATE(counter)->count = 0;
p = PRIVATE(counter)->waitingList;
q = p->next;
while (q->next != NULL) {
    p->next = q->next;
    returnOK = CloseHandle(q->signal);
    CHECKOTHER(returnOK == TRUE);
    free(q);
    q = p->next;
}

return STHREADS_ERROR_NONE;
}

/*-----*/

#undef PRIVATE

/*-----*/
/* Locks */
/*-----*/

typedef struct {
    int initialized, finalized;
    HANDLE holder;
    CRITICAL_SECTION lock;
} PrivateLock;

#define PRIVATE(lockPtr) ((PrivateLock *) (lockPtr))

/*-----*/

int SthreadsLockInitialize(SthreadsLock *lock)
{
    CHECKINPUTVALUE(lock != NULL);
    PRIVATE(lock)->initialized = INITIALIZED;
    PRIVATE(lock)->finalized = ~FINALIZED;
    PRIVATE(lock)->holder = NULL;
    InitializeCriticalSection((LPCRITICAL_SECTION) &PRIVATE(lock)->lock);
    return STHREADS_ERROR_NONE;
}

/*-----*/

int SthreadsLockFinalize(SthreadsLock *lock)
{
    CHECKINPUTVALUE(lock != NULL);
    CHECKUNINITIALIZED(PRIVATE(lock)->initialized == INITIALIZED);
    CHECKFINALIZED(PRIVATE(lock)->finalized == ~FINALIZED);
    CHECKINUSE(PRIVATE(lock)->holder == NULL);

    PRIVATE(lock)->finalized = FINALIZED;
    DeleteCriticalSection((LPCRITICAL_SECTION) &PRIVATE(lock)->lock);

    return STHREADS_ERROR_NONE;
}

/*-----*/

int SthreadsLockAcquire(SthreadsLock *lock)
{
    HANDLE thisThread;

    thisThread = GetCurrentThread();
    believe(thisThread!= NULL);

```

```

CHECKINPUTVALUE(lock != NULL);
CHECKUNINITIALIZED(PRIVATE(lock)->initialized == INITIALIZED);
CHECKFINALIZED(PRIVATE(lock)->finalized == ~FINALIZED);

EnterCriticalSection((LPCRITICAL_SECTION) &PRIVATE(lock)->lock);
believe(PRIVATE(lock)->holder == NULL ||
        PRIVATE(lock)->holder == thisThread);
CHECKLOCKHELD(PRIVATE(lock)->holder == NULL);
PRIVATE(lock)->holder = thisThread;

return STHREADS_ERROR_NONE;
}

/*-----*/

int SthreadsLockRelease(SthreadsLock *lock)
{
    HANDLE thisThread;

    thisThread = GetCurrentThread();
    believe(thisThread != NULL);

    CHECKINPUTVALUE(lock != NULL);
    CHECKUNINITIALIZED(PRIVATE(lock)->initialized == INITIALIZED);
    CHECKFINALIZED(PRIVATE(lock)->finalized == ~FINALIZED);
    CHECKLOCKNOTHELD(PRIVATE(lock)->holder == thisThread);

    PRIVATE(lock)->holder = NULL;
    LeaveCriticalSection((LPCRITICAL_SECTION) &PRIVATE(lock)->lock);

    return STHREADS_ERROR_NONE;
}

/*-----*/

#undef PRIVATE

/*-----*/
/* Barriers */
/*-----*/

typedef struct {
    int initialized, finalized;
    int numThreads;
    int numWaiting;
    HANDLE gate[2];
    int currentGate; /* 0 or 1 */
    CRITICAL_SECTION lock;
} PrivateBarrier;

#define PRIVATE(barrierPtr) ((PrivateBarrier *) (barrierPtr))

/*-----*/

int SthreadsBarrierInitialize(SthreadsBarrier *barrier, int numThreads)
{
    CHECKINPUTVALUE(barrier != NULL);
    CHECKINPUTVALUE(numThreads >= 1);

    PRIVATE(barrier)->initialized = INITIALIZED;
    PRIVATE(barrier)->finalized = ~FINALIZED;
    PRIVATE(barrier)->numThreads = numThreads;
    PRIVATE(barrier)->numWaiting = 0;
    PRIVATE(barrier)->gate[0] = CreateEvent(NULL, TRUE, FALSE, NULL);
    CHECKSYNCCREATE(PRIVATE(barrier)->gate[0] != NULL);
    PRIVATE(barrier)->gate[1] = CreateEvent(NULL, TRUE, TRUE, NULL);
    CHECKSYNCCREATE(PRIVATE(barrier)->gate[1] != NULL);
    PRIVATE(barrier)->currentGate = 0;
    InitializeCriticalSection((LPCRITICAL_SECTION) &PRIVATE(barrier)->lock);

    return STHREADS_ERROR_NONE;
}

/*-----*/

```

```

int SthreadsBarrierFinalize(SthreadsBarrier *barrier)
{
    BOOL returnOK;

    CHECKINPUTVALUE(barrier != NULL);
    CHECKUNINITIALIZED(PRIVATE(barrier)->initialized == INITIALIZED);
    CHECKFINALIZED(PRIVATE(barrier)->finalized == ~FINALIZED);
    CHECKINUSE(PRIVATE(barrier)->numWaiting == 0);

    PRIVATE(barrier)->finalized = FINALIZED;
    returnOK = CloseHandle(PRIVATE(barrier)->gate[0]);
    CHECKOTHER(returnOK == TRUE);
    returnOK = CloseHandle(PRIVATE(barrier)->gate[1]);
    CHECKOTHER(returnOK == TRUE);
    DeleteCriticalSection((LPCRITICAL_SECTION) &PRIVATE(barrier)->lock);

    return STHREADS_ERROR_NONE;
}

/*-----*/

int SthreadsBarrierPass(SthreadsBarrier *barrier)
{
    int currentGate, nextGate;
    BOOL returnOK;
    DWORD returnCode;

    CHECKINPUTVALUE(barrier != NULL);
    CHECKUNINITIALIZED(PRIVATE(barrier)->initialized == INITIALIZED);
    CHECKFINALIZED(PRIVATE(barrier)->finalized == ~FINALIZED);

    EnterCriticalSection((LPCRITICAL_SECTION) &PRIVATE(barrier)->lock);
    currentGate = PRIVATE(barrier)->currentGate;
    PRIVATE(barrier)->numWaiting = PRIVATE(barrier)->numWaiting + 1;
    if (PRIVATE(barrier)->numWaiting == PRIVATE(barrier)->numThreads) {
        nextGate = (currentGate + 1)%2;
        returnOK = ResetEvent(PRIVATE(barrier)->gate[nextGate]);
        CHECKOTHER(returnOK);
        PRIVATE(barrier)->numWaiting = 0;
        returnOK = SetEvent(PRIVATE(barrier)->gate[currentGate]);
        CHECKOTHER(returnOK);
        PRIVATE(barrier)->currentGate = nextGate;
        LeaveCriticalSection((LPCRITICAL_SECTION) &PRIVATE(barrier)->lock);
    } else {
        LeaveCriticalSection((LPCRITICAL_SECTION) &PRIVATE(barrier)->lock);
        returnCode = WaitForSingleObject(
            PRIVATE(barrier)->gate[currentGate], INFINITE);
        CHECKOTHER(returnCode != WAIT_FAILED);
    }

    return STHREADS_ERROR_NONE;
}

/*-----*/

int SthreadsBarrierReset(SthreadsBarrier *barrier, int numThreads)
{
    BOOL returnOK;

    CHECKINPUTVALUE(barrier != NULL);
    CHECKUNINITIALIZED(PRIVATE(barrier)->initialized == INITIALIZED);
    CHECKFINALIZED(PRIVATE(barrier)->finalized == ~FINALIZED);
    CHECKINUSE(PRIVATE(barrier)->numWaiting == 0);
    CHECKINPUTVALUE(numThreads >= 1);

    PRIVATE(barrier)->numThreads = numThreads;
    PRIVATE(barrier)->numWaiting = 0;
    returnOK = ResetEvent(PRIVATE(barrier)->gate[0]);
    CHECKOTHER(returnOK);
    returnOK = SetEvent(PRIVATE(barrier)->gate[1]);
    CHECKOTHER(returnOK);
    PRIVATE(barrier)->currentGate = 0;
}

```

```

    return STHREADS_ERROR_NONE;
}

/*-----*/

#undef PRIVATE

/*-----*/
/* Priorities */
/*-----*/

int SthreadsGetCurrentPriority(int *priority)
{
    HANDLE currentThread;
    int currentPriority;

    CHECKINPUTVALUE(priority != NULL);

    currentThread = GetCurrentThread();
    believe(currentThread != NULL);
    currentPriority = GetThreadPriority(currentThread);
    believe(currentPriority != THREAD_PRIORITY_ERROR_RETURN);

    *priority = currentPriority;

    return STHREADS_ERROR_NONE;
}

/*-----*/

int SthreadsSetCurrentPriority(int priority)
{
    HANDLE currentThread;
    BOOL returnOK;

    CHECKINPUTVALUE(ValidPriority(priority));

    currentThread = GetCurrentThread();
    believe(currentThread != NULL);
    returnOK = SetThreadPriority(currentThread, priority);
    believe(returnOK);

    return STHREADS_ERROR_NONE;
}

/*-----*/

```